1976 CONFERENCE ON PARALLEL PROCESSING Enslow, Ed. 76CH1127-0C

PROCEEDINGS OF THE 1976 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

PHILIP H. ENSLOW JR.
Editor

Cosponsored by the

Wayne State University

and the

IEEE Computer Society

IEEE Catalog No. 76CH1127-0C

PROCEEDINGS OF THE 1976 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

PHILIP H. ENSLOW JR.
Editor

Papers presented on
August 24-27, 1976

Co-Sponsored by

Department of Electrical and Computer Engineering
WAYNE STATE UNIVERSITY
Detroit, Michigan

and the

25 YEARS OF SERVICE

IEEE Computer Society

In Cooperation with the

acm

Association for Computing Machinery

PREFACE

This series of conferences on parallel processing has matured into a truly international event, and I am extremely pleased to have been associated with the 1976 meeting. What started as a very small meeting on a special aspect of parallel processing in 1972 has expanded into a major meeting covering every facet of the subject with contributions from all over the world. The series continues under the general chairmanship of Professor Yse-yun Feng, and I was honored to be able to share some of the work of organizing the 1976 International Conference on Parallel Processing. This year the meeting had the formal support of both the IEEE Computer Society and the Association for Computing Machinery. This support is gratefully acknowledged, especially that of the Computer Society which is handling the production and distribution of these Proceedings and that of SIGARCH for assistance in organizing a session.

All of the papers submitted to this conference were formally reviewed, and I would like to sincerely thank the 59 individuals who served as referees for the more than 80 papers submitted. The efforts of these workers, who are identified at the end of these Proceedings, were essential in organizing a quality meeting. The workload was heavier this year than ever before, but I believe that you will agree with me that they did an outstanding job. I would also like to call your attention to a new feature of this year's meeting and acknowledge the outstanding efforts of the winners of the awards for the Best Presentation and the Best Paper. These individuals, identified on the next page, were selected by the best jury possible, the attendees at the conference.

It has been my pleasure to work on this conference, and I look forward with high anticipation to the 1977 meeting.

Philip H. Enslow, Jr.
Program Chairman

SPECIAL AWARDS

1976 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

Best Presentation

Professor David S. Wise
Indiana University

"The Impact of Applicative Programming on Multiprocessing"
by Daniel P. Friedman and David S. Wise.

Best Paper

Professor Jerome Rothstein
Ohio State University

"On the Ultimate Limitations of Parallel Processing"

TABLE OF CONTENTS

Page

KEYNOTE ADDRESS
Parallel Processing: Commercial Needs, Trends, and Perspectives
Dr. Robert R. Johnson
Burroughs Corporation

SESSION 1 APPLICATIONS: IMAGE PROCESSING
Chairperson: Professor J. Rothstein

Highly Parallel Digitized Geometric Transformations Without Matrix Multiplication
C. F. R. Weiman ... 1

Enhancement of Computing Power In Multiprocessor Systems For Processing of Digitized Pictures
K. Vorgrimler ... 11

Application Of Distributed Processing To The Production Of Digital Terrain Data
Dennis E. Moellman and Robert A. Meyer ... 18

Application Of A Parallel Processing Computer In LACIE
Sherwin Ruben, John Lyon, Rudolf Faiss and Matthew Quinn ... 24

High-Resolution Image Processing On A Parallel Computer System
W. W. Gaertner, M. P. Patel, S. S. Reddi, C. T. Retter and I. M. Singh ... 33

An Implementation Of The Hadamard Transform On The Staran Associative Array Processor
Annette J. Krygiel ... 34

SESSION 2: OPERATING SYSTEMS
Chairperson: Lieutenant Colonel F. J. Hilbing

On The Formal Definition Of Processes
Pamela Zave ... 35

Hierarchical Monitors
A. M. Lister and P. J. Sayer ... 43

Garbage Collection With Multiple Processes: An Exercise In Parallelism
Leslie Lamport ... 50

Hierarchial Properties of Concurrency
G. S. Tjaden ... 55

SESSION 3: SYSTEM ARCHITECTURE AND ORGANIZATION I
Chairperson: M. Kesselman

The Flip Network In Staran
Kenneth E. Batcher ... 65

Construction Of A Versatile Data Manipulator For Parallel/Associative Processors
W. W. Gaertner, M. P. Patel, C. T. Retter and I. M. Singh ... 72

TABLE OF CONTENTS (CONT'D)

Page

SESSION 4: SCHEDULING
Chairperson: Dr. C. P. Hsieh

Fast Algorithms For Bounding The Performance Of Multiprocessor Systems
Chao-Chih Yang 73

Two Models Of Task Overlap Within Jobs Of Multiprocessing Multiprogramming Systems
Mamoru Maekawa and Donold L. Boyd 83

Optimal Scheduling Of Vector Computations In A Reconfigurable Shared-Resource Array Processing System
Alexander Thomasian and Algirdas Avizienis 92

SESSION 5: SYSTEM PERFORMANCE I
Chairperson: Joe McKay

A Model For A Shared Resource Multiprocessor
Lawrence S. Cheung and Frederic J. Mowle 93

Performance Analysis Of A Data-Flow Processor
David P. Misunas 100

An Analytic Approach To Performance Analysis For A Class Of Data Flow Processors
Susan C. Meyer 106

On The Evaluation Of Array Computers
R. Hemmersbach and D. Schutt 116

SESSION 6: RELIABILITY
Chairperson: Captain R. A. Johnson

Error Detection And Recovery In A Data-Flow Computer
David P. Misunas 117

Reliability Aspects Of The Illiac IV Computer
Iftikhar A. Baqai and Tomas Lang 123

Computer Architectures For Advanced Air Traffic Control Applications
Andres Zellweger 132

SESSION 7: APPLICATIONS
Chairperson: Dr. W. W. Gaertner

Distributed Processing For Signal Processor Using The Building Block Signal Processor
Frank P. Hiner III 140

Associative-Parallel Applications to Radar Signal Processing
K. L. Schaffer 145

A Receiver For PCM Coded Digitone And MF Signals Using Associative Processing
Eugene S. Y. Shew and Jack M. Cotton 154

TABLE OF CONTENTS (CONT'D.)

Page

Radar Data Processing on the Alap
Hubert H. Love 161

Architecture and Simulation of an Associative Processor Integrated Circuit
Jackylene Hood, Maitang Mark, and Jack Cotton 168

Application of Pepe to Real-Time Digital Filtering
D. B. Kimsey, L. E. Hand, and H. T. Nagle, Jr. 169

SESSION 8: STARAN AND RELATED TOPICS
Chairperson: Oscar Reimann

High Level Language for Associative and Parallel Computation with Staran
R. G. Lange 170

Analysis of the Awacs Passive Tracking Algorithms on the Radcap Staran
Robert Katz 177

Automatic Track Initiation Using the Radcap Staran
Edward C. Stanke, II 187

Concept for a Computer Architecture Research Facility
A. Klayton 189

A Content-Addressed Memory Designed for Data Base Applications
George A. Anderson and Richard Y. Kain 191

SESSION 9: PERFORMANCE II
Chairperson: Dr. L. Lamport

Time and Parallel Processor Bounds for Linear Recurrence Systems with Constant Coefficients
S. C. Chen 196

On the Ultimate Limitations of Parallel Processing
Jerome Rothstein 206

SESSION 10: MULTIPLE-MICROPROCESSORS
Chairperson: David Freedman

An Efficient Multiprocessor Architecture
Vincent 213

Design Considerations in Multi-Minicomputer Performance
Tadaaki Bandoh and Yukio Kawamoto 219

A Modular Vector Processing Unit
S. R. Ahuja and J. R. Jump 220

A Shared Memory Technique for Different Microprocessors
Ronald L. Krutz and Bob Reynouard 221

TABLE OF CONTENTS (CONT'D.)

Page

SESSION 11: FUNDAMENTAL THEORY
Chairperson: Professor H. M. Yamada

Parallel Recognition of Patterns: Insights from Formal Language Theory
Michael Moshell and Jerome Rothstein 222

Some Computational and System Theoretic Properties of Regular Processor Networks
Renato M. Ermann and William I. Grosky 230

A Proof Method for Cyclic Programs
Nissim Francez and Amir Pnueli 235

On Determinacy and Equivalence of Parallel Program Schemata
Manilal Daya 246

SESSION 12: LANGUAGE ISSUES
Chairperson: Professor Roy Zingg

Coordination of Parallel Processes in PL/1
Howard S. Modell, Ronnie G. Ward and Ted M. Sparr

On the Time Required to Parse an Arithmetic Expression for Parallel Processing
Ross A. Towle and Richard P. Brent 254

Lau System Software: A High Level Data Driven Language for Parallel Programming
O. Gelly, et.al 255

A High Level Language Oriented Multiprocessor
Mario F. de La Guardia and James A. Field 256

The Impact of Applicative Programming on Multiprocessing
Daniel P. Friedman and David S. Wise 263

SESSION 13: SYSTEM ARCHITECTURE AND ORGANIZATION II
Chairperson: John Cornell

Single Instruction Stream-Multiple Data Stream Machine Interconnection Network Design
Howard Jay Siegel 273

Effectiveness of Some Processor/Memory Interconnections
K. Y. Wen and D. H. Lawrie 283

Lau System Architecture: A Parallel Data-Driven Processor Based on Single Assignment
A. Plas, et.al. 293

Upper Bounds on the Performance of Some Processor-Memory Interconnections
R. C. Pearce and J. C. Majithia 303

TABLE OF CONTENTS (CONT'D.)

Page

SESSION 14: SCHEDULING II
Chairperson: Professor David Kuck

Operating System Modelled as a Conglomerate of Interdependent Activities
T. Fend and C. P. Hsieh 304

A Scheme for the Parallel Execution of Sequential Programs
C. V. Ramamoorthy and W. H. Leung 312

On Further Applications of the Hu Algorithm to Scheduling Problems
Edgar Nett 317

HIGHLY PARALLEL DIGITIZED GEOMETRIC TRANSFORMATIONS[a] WITHOUT MATRIX MULTIPLICATION

Carl F. R. Weiman
Mathematical and Computing Sciences Dept.
Old Dominion University
P.O. Box 6173
Norfolk, Virginia 23508

Abstract -- A new, computationally simple, highly parallel method for performing linear geometric transformations on digitized pictures is presented. Matrix multiplication is avoided by using a weighting scheme. Grid digitization is fundamental to the computation rather than being an undesirable source of error as in conventional methods. Microprocessor implementation based on vector parallelism suggests the possibility of real time animation of grey-scale pictures. Applications are not restricted to computer graphics and image processing, however, but are general to any system in which coordinates and linearity are involved, e.g., the numerical solution of PDE's. The transformation method is based on an interpretation of Rothstein's straight line code as an operator for digitized linear interpolation rather than as the description of a geometric figure.

I. Introduction

This paper describes a new, computationally simple and highly parallel method for performing affine transformations on digitized pictures and similar grid-based systems. Geometrically, affine transformations map parallelograms into parallelograms and are involved in linear axis scaling, shearing, and rotation. In picture processing, these transformations are useful in achieving picture registration for comparison, recognition, or mosaicing with other pictures; in computer graphics successive transformations yield animation. Ordinarily, an affine transformation is applied by multiplying all point coordinate tuples by a constant matrix. In transforming grey-scale pictures the original and transformed digitization grids generate 2-D moiré patterns of holes which must be filled by smoothing, thereby destroying information [1]. In the method presented here, matrices are not used and no operations more complex than addition of integers are needed. Grid digitization of the picture is fundamentally involved in the computation rather than being an undesirable source of error. The method is most suitable for parallel implementation on vector processors; real-time animation of arbitrarily rich gray-level pictures is straightforward using present technology.

Applications are not restricted to computer graphics and image processing but general to any system in which coordinates and linearity are involved, for example in the numerical solution of partial differential equations. The method is based on an interpretation of Rothstein's digitized straight line code [2] as a rule or operator for digitized linear interpolation rather than as representing a geometric entity.

[a] Research partly supported under ONR grant N00014-75-C-0571. Computation partly supported by ERDA under contract E (11-1) - 3077. Both at Courant Institute, New York University while author was Associate Research Scientist in Computer Science Department.

II. Rothstein's Code for Digitized Straight Lines

Rothstein's code is a binary sequence, each of whose digits corresponds to the nearest neighbor configuration of a grid cell crossed by a straight line; 0 corresponds to a cell whose neighbors on opposite sides are crossed by the line and 1 to a cell whose neighbors are crossed on adjacent sides (see figure 1). In the latter case the next cell is ignored, avoiding redundancy and yielding one code digit per grid column (grid row for slopes whose absolute values exceed unity). For a line of slope p/q where $0 \leq p \leq q$ are integers with no common factors, the code has period q with p 1's per period. The digit sequence can be simply generated without solving the equation of the straight line at intersections with grid parallels by viewing the line between (0,0) and (q,p) as divided into pq equal segments and noticing that a digit occurs once for each interval of p such segments (i.e., the distance between two successive grid verticals). That digit is 1 if the interval in question also happens to contain the termination of an interval of q such segments (i.e., the line crosses a grid horizontal); otherwise, the digit is 0. This can be expressed in hardware (figure 2a) by synchronizing to the same clock, two cyclic binary shift registers of lengths p and q respectively, detecting end-around shifts of a single bit in each to determine code digits. A faster method using more hardware consists of successively adding p to a modulo q counter and detecting values less than p to generate code 1's (figure 2b). The latter can also be expressed as a generating function:

$$f = e^{2\Pi i(p/q)n} \qquad (i=\sqrt{-1})$$

The nth code digit is 1 for $\mathrm{Arg}(f) < 2\Pi(p/q)$ and 0 otherwise.

III. Digitized Affine Transformations

A. Axis Scaling. The geometry of figure 1 shows that the code comprises the most homogeneous possible distribution of p 1's among q digits. This suggests scaling the x-axis of a gray-scale picture by a ratio of q to p by distributing the p columns of the original picture among q of the transformed picture using the same homogeneous distribution. This is a digitized approximation to the affine transformation

$$(x,y)\begin{pmatrix} q/p & 0 \\ 0 & 1 \end{pmatrix} = (x\cdot q/p, y) \ ,$$

subject to the constraint that the picture grid cannot change. Shrinking the picture along the x-axis (replace q/p with p/q above) similarly corresponds to selecting p columns from q of the original picture according to the same homogeneous distribution. Figure 3 illustrates this technique with the corresponding code written above or below appropriate columns; the number within each cell represents a gray level. Unfortunately, in the case of expansion, empty "seams" are introduced and in contraction, columns are deleted. Such artifacts could be reduced by spatial smoothing, an undesirable solution not only because information is lost but also because gap geometry may "moiré" with picture features. This strong dependence on the relative positions of the grid and picture violates intuitions about picture invariance under translation.

Looking again at figure 2, note that changing the relative phases of the shift registers permutes the resulting code digits cyclically but does not change the average density of code 1's nor the homogeneity of their distribution. Thus, starting the code at any position other than when both registers are at the zero position yields a column selection rule equally as good in terms of homogeneity. Averaging the gray-levels resulting from all cyclic permutations of the column selection code therefore averages gap positions over all columns, eliminating discontinuities. No parts of the picture are selectively altered because all cells are represented. Figure 4 illustrates this averaging process for a ratio of 4/3.

Though the averaging process just described satisfies informational intuitions, it must be proven geometrically correct. That is, the resulting grey-scale picture must be the same as would have resulted from optically scaling the original picture and then redigitizing. The proof requires some results from the geometry of numbers beyond the scope of this paper but covered in detail in [3]. The outline of the proof follows. Stretching a picture in the continuous (non-digitized) case by the factor q/p can be viewed as a perspectivity through a point at infinity which projects p consecutive originally unit width columns of the original picture onto q consecutive unit width columns of the transformed picture; the code for p/q is a description of where column boundaries fall in the image. Each of the p original columns spreads into several of the q columns; the relative contribution of each original column to each new column is proportional to the relative area of the stretched image of the former occupying the new column in question. Now consider figure 1 as a cross-section of the columns in the obvious sense. Relative area in the preceding sentence becomes relative length under this interpretation. These lengths could be measured by stepping along the q-cells 1/p units at a time counting steps and observing when the image of a p-cell boundary is crossed. Since step lengths are equal, each unit distance is equivalent to a count of p. If this stepping proceeds from each of the p-cell boundary images, q steps are both necessary and sufficient to count the lengths. But this yields precisely the same result as translating a line of slope p/q vertically by one grid cell and noting the number of times a 1 appears in each column. Since 1's change position only when the line crosses lattice points, and between such lattice points the code must be identical to the original, the result of translation must be a sequence of cyclic shifts in the code. That this sequence consists precisely of all possible shifts is also proved in [3].

Visual corroboration of the averaging process is illustrated in figure 5a. A mathematically defined test pattern was stretched horizontally using the code averaging algorithm just described. The stretch ratio is approximately the ratio of row to column spacing on the printer and was chosen to correct this distortion caused by erroneously assuming equal spacing. Note that edges in the figure appear to have slopes ± 1 but a detailed examination reveals a boundary path related to the straight line code. Figure 6 was derived from a digitized photograph of a cat. Line code stretching algorithms applied horizontally and vertically yielded figure 7, illustrating the smoothness of the process even in delicately shaded retions.

B. <u>Shearing Transformations</u>. Combinations of horizontal and vertical stretching map rectangles into rectangles without altering the direction of edges. Shearing transformations, characterized by matrices of the form

$$\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}^T \text{(horizontal) and } \begin{pmatrix} 1 & 0 \\ b & 1 \end{pmatrix}^T \text{(vertical)}$$

map rectangles into parallelograms, altering the direction of one set of parallel edges. They are of interest here not only because they can be easily carried out using an averaging method similar to that just described for axis scaling, but also because appropriate combinations of shearing and scaling yield the entire group of affine transformations.

Using the same reasoning as for scaling, the code for p/q can be regarded as a rule for shearing the grid upward by sliding a column and those to its right upward one unit whenever a code digit 1 appears under a column. Just as in scaling, the jagged steps are removed by averaging over all possible cyclic shifts in the code position. For horizontal shearing, the word "column" should be replaced by "row", and "under" by "next to" Figure5b illustrates the application of a shearing transformation effected by digitized averaging.

The transformations just described which average gray levels according to rules derived from the straight line code are henceforth referred to as digitized affine transformations (DATs) to distinguish them from the ordinary continuous affine transformations (CATs). The way DATs were presented, the computation involves moving columns of the original picture to several different positions in the grid of its transformed image. This computation can be rearranged by regarding each column of the image picture as having contributions from several columns of the original. The weight of each contribution is simply the fraction of the time the code digit 1, corresponding to a column in the original, spends in the column of the image when the code is cyclically permuted. This weighting scheme resembles a digitized version of a filtering or smoothing convolution. An important difference is that here the weighting coefficients are slightly different for each column in a period of code. However, they need only be calculated once for the entire picture by counting code 1's in columns throughout one cyclic permutation of the code. Viewing the computation as a weighting scheme or "pseudo-convolution" has important consequences for parallel implementation as will be discussed in the overview. The code has excellent approximation properties related to continued fractions [2], [3] which permit the use of shorter periods than might be expected to yield accurate results. In particular, accuracy to 1/q th of a grid unit is achieved by codes of period q. Shears and stretches by irrational amounts may thus be approximated to any desired accuracy.

C. Rotation Through Arbitrary Angles. Rotation of a digitized picture through an arbitrary angle is computationally complex using conventional matrix multiplication methods [1]. It is nevertheless important for several reasons. At least one direction which is invariant under each DAT described above must be parallel to a grid axis. Shearing and stretching in an arbitrary direction cannot be accomplished by applying such transformations. However, if composition with an arbitrary rotation were possible, this directional constraint would be relaxed. Then, choice of coordinate directions is arbitrary, a necessary property of any general geometric system [4]. In picture processing applications, rotation is vital to picture registration; in computer graphics it is important for non-trivial animation. Fortunately, arbitrary rotation in the continuous case can be decomposed into shearing and scaling CAT's with the same special orientations as the DAT's presented above. Replacing these CAT's with the corresponding DAT's yields arbitrary rotation of digitized pictures using DAT's only. The accuracy of the result is as good as that of the shearing and scaling DAT's. Details follow.

Consider a unit square with one corner at the origin and a side in the fourth quadrant making an internal angle $-\Theta$ with the x-axis. (Refer to figure 8a). Applying the shear

$$\begin{pmatrix} 1 & \tan\Theta \\ 0 & 1 \end{pmatrix}$$

yields a parallelogram with one pair of sides parallel to the x-axis (figure 8b). Next applying

$$\begin{pmatrix} 1 & 0 \\ -\sin\Theta\cos\Theta & 1 \end{pmatrix}$$

yields a rectangle with pairs of sides parallel to x and y axes (figure 8c). Applying the scaling transformations

$$\begin{pmatrix} 1 & 0 \\ 0 & \cos\Theta \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \frac{1}{\cos\Theta} & 0 \\ 0 & 1 \end{pmatrix}$$

then yields a unit square (figure 8d). Clearly the result is a rotation by Θ degrees since the product of the transformations is

$$\begin{pmatrix} \cos\Theta & \sin\Theta \\ -\sin\Theta & \cos\Theta \end{pmatrix}$$

The digitized versions of these transformations can be applied by using the codes corresponding to the non-integer quantities in the matrices above, approximated to any desired accuracy. At first sight, this approximation appears to involve several trigonometric calculations, but if an angle is specified by giving the direction (q,p) of the line making that angle with the x-axis, no trigonometric calculations are necessary. The tangent is p/q and the code can be generated directly using the schemes illustrated in figures 2 a and b. The sine and cosine can be derived from pythagorean relationships with $\sqrt{p^2 + q^2}$. Taking reciprocals requires no calculation; one simply interchanges stretching and shrinking. Though these calculations are more complex than addition, they involve small integers and need only be performed once, i.e., to generate four codes, regardless of the number of cells in the grid.

IV. Overview of Geometric and Complexity Considerations in a Parallel Computation Schema

Geometrically, DAT's appear to be a valid alternative to matrix multiplication for applying linear transformations. In sequential implementations such as those which generated the illustrations just presented, however, each matrix operation is replaced by several additions, somewhat offsetting the possible advantage of algorithm simplicity by increasing computation time. The following vector oriented parallel computation schema exploits DAT characteristics to permit speedups by several orders of magnitude. The design depends on the fact that all DAT's are characterized by the summation of grey values from cells in a restricted neighborhood which is oriented in the same direction (either horizontal or vertical) for all cells on the grid. Thus, computations for cells which are neighbors perpendicular to this direction can be carried out simultaneously without interaction. Consider the example the stretching process illustrated in figure 4. Summation neighborhoods are horizontally oriented; e.g., the grey value of any cell in column three in the new picture is the average of values from columns three and two (the latter twice) in the original picture. This same weighting rule could be applied to all cells in that column in a single step. The design thus calls for a vector of identical accumulators, each vector component representing a cell in the column of the new picture. Computational capabilities required include addressing a small number of neighboring vectors and adding them up (preserving vector component independence). As the vector of accumulators looks at successive columns, neighborhood shape (weighting rule) changes according to the straight line code for the stretching ratio. For example, in figure 4 column four of the new picture receives contributions only from column three of the original picture.

A reasonably high resolution picture consisting of 1000 x 1000 pixels would require a processing vector of length 1000. Factors vitally important to practical LSI implementation are identical structure and weighting rules for each vector component (redundancy) and complete avoidance of thousand-fold construction of either multiplication hardware or interconnection between components [5]. The thousand-fold speedup afforded by vector organization could reduce computation time from a half-minute per frame (i.e., thirty microseconds, per pixel) to thirty milliseconds, well within the range of real-time animation of arbitrarily rich grey-scale pictures consisting of a million pixels. Such speeds are inconceivable with current computer graphics techniques which matrix multiply coordinate lists. Similar transformation speeds are essential (and achievable) for real-time robot visual perception. The weighted averaging capabilities of this design can also be used to implement smoothing, edge detection and enhancement, and other conventional local picture processing operations at high speed.

A variety of potential applications far broader than computer graphics and image processing are possible because DAT's are the basis of a new kind of computational geometry which converges to affine Euclidean geometry. This digitized geometry is well matched to the discrete nature of digital computation and should be useful in many areas of applied mathematics. For example, in the numerical solution of partial differential equations by relaxation on uniform grids, linear changes of coordinates can be rapidly effected without redefining the grid. This might be useful when boundary geometry or wave propagation directions are approximated by straight line paths making arbitrary angles with grid parallels. Many promising theoretical areas of investigation are suggested as well. These include extension to digitized projective geometry, curvilinear coordinates, higher dimensional spaces, and many other areas traditionally described in terms of continuity. There are some fundamental differences between the continuous approach and the digitized approach presented here. One example is the use of lines (digitized) rather than points as the fundamental objects manipulated by computation. Though duality between these elements is well known [4] applications conventionally involve point manipulation. With DAT's, the line codes correspond to scan paths in arbitrary directions whose discontinuities resulting from digitization are smoothed by averaging (pseudo-convolution). The quantized geometry of the grid yields an "uncertainty principle" which corresponds to the inability to localize points in any neighborhood smaller than a grid cell, an inherent constant of the system. While this would be intolerable in continuous geometry, it simplifies the computations of our "transformational" geometry using DAT's.

Acknowledgements

Professor Rothstein embodied elements of geometry, number theory, approximation theory, information theory, ergodic theory, and theory of formal languages in a clear notation in the design of the straight line code. The DAT application presented here is one of many possible applications. I am deeply indebted to him for the code and exploration of related worlds during and after my Ph.D. dissertation research under him.

I am also indebted to Bernard Shunfenthal who as a student of mine at New York University converted my conjectures about DAT's into computer programs for generating and transforming greyscale overprint pictures. These not only visually confirmed the feasibility of the method but gave me a great deal of intellectual and aesthetic satisfaction. Some of the products of his work are shown in the figures illustrating transformations of test patterns. The cat is an overprint version of a digitized picture of Sam Harbison's cat "Schopenhauer". The deck was provided to me by Leonard Vanek, formerly associated with the Princeton picture processing group but now at The Courant Institute. The blow-up of the eye was executed by Hartsell Teagle at Old Dominion University.

References

[1] E. G. Johnston and A. Rosenfeld, "Geometrical Operations on Digitized Pictures" in B. Lipkin and A. Rosenfeld (eds.), Picture Processing and Psychopictorics, Academic Press, New York, (1970), pp. 217-240.

[2] J. Rothstein and C. F. R. Weiman, "Parallel and Sequential Specification of a Context Sensitive Language for Straight Lines on Grids", Computer Graphics and Image Processing, Vol. 5, (March, 1976), pp. 106-124.

[3] C. F. R. Weiman and J. Rothstein, Pattern Recognition by Retina-Like Devices, Computer and Information Science Dept., Ohio State University, OSU-CISRC-TR-72-8,

(AD 214 665/2), (1972), 154 pp.

[4] A. Tuller, A Modern Introduction to Geometries, van Nostrand, New York, 1967, 201 pp.

[5] B. R. Borgerson, "The Viability of Multimicroprocessor Systems", Computer, (IEEE Computer Society), Vol. 9, (January, 1976), pp. 26-30.

[6] C. F. R. Weiman and J. Rothstein, "Polyautomaton Design for Recognizing Certain L System Languages by Parallel Computation", in Proceedings of 1975 Sagamore Computer Conference on Parallel Processing, Syracuse University and IEEE Computer Society (1975), pp. 168-170.

[7] E. Artzy, J. A. Hinds, and H. J. Saal, "A Fast Division Technique for Constant Divisors", Comm. of the ACM, Vol. 19, (February 1976), pp. 98-101.

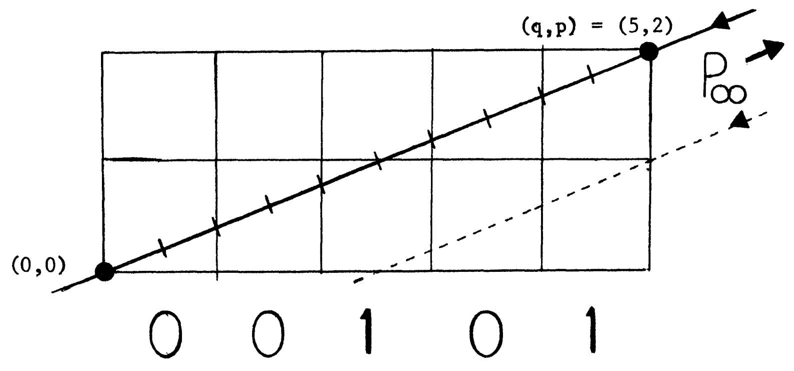


Figure 1: Rothstein's Code for Slope 2/5.

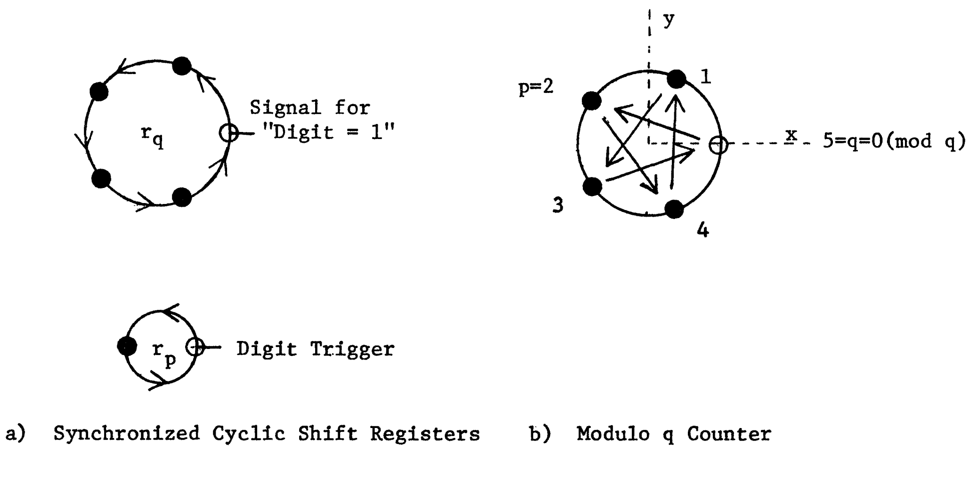


Figure 2: Code Generating Hardware

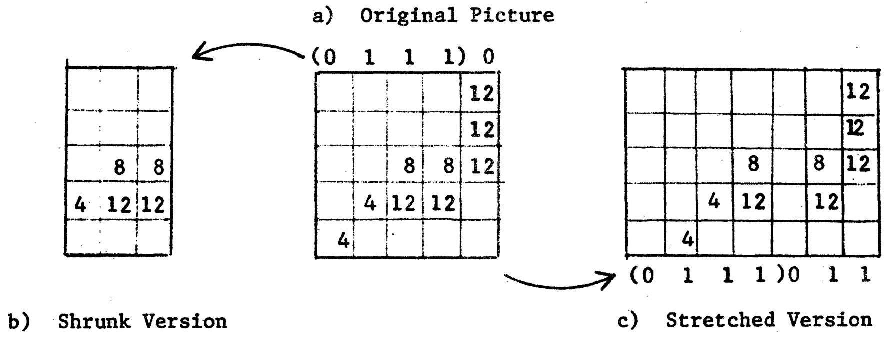


Figure 3: Digitized Stretching and Shrinking

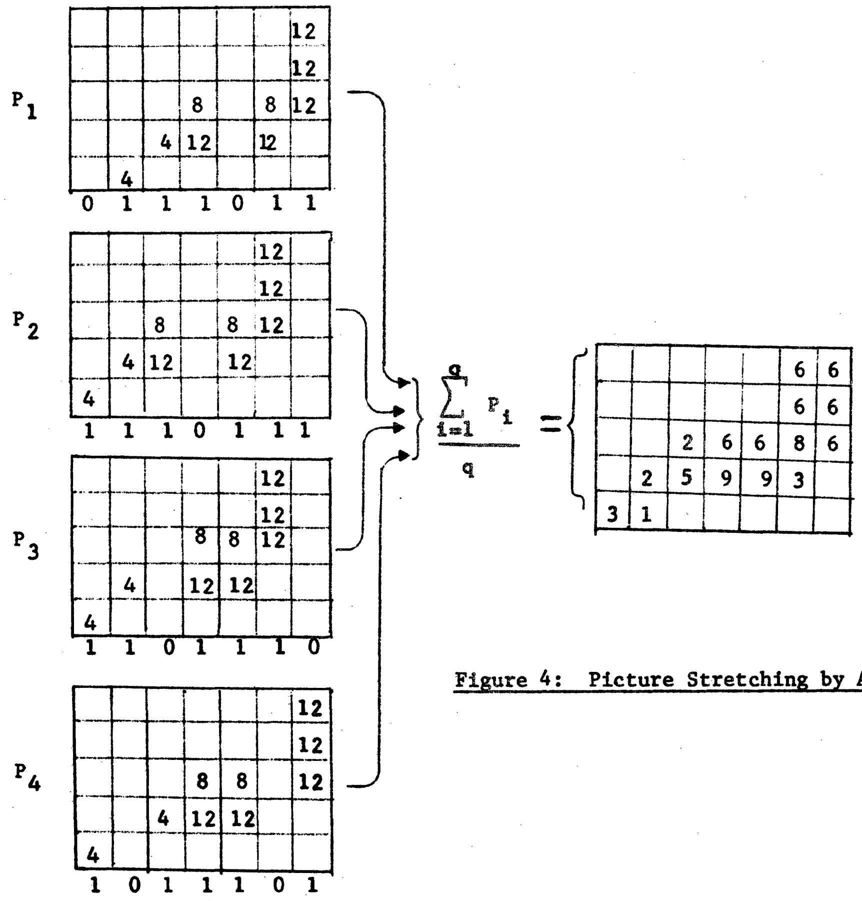


Figure 4: Picture Stretching by Averaging

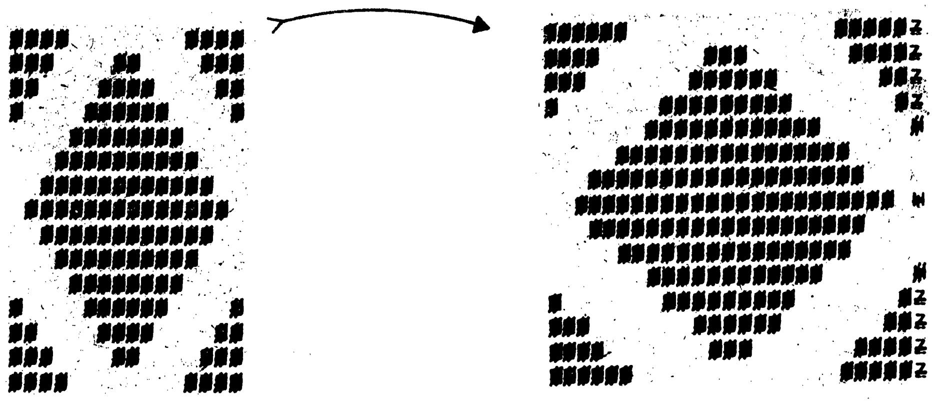


Figure 5a: Illustration of Digitized Stretching

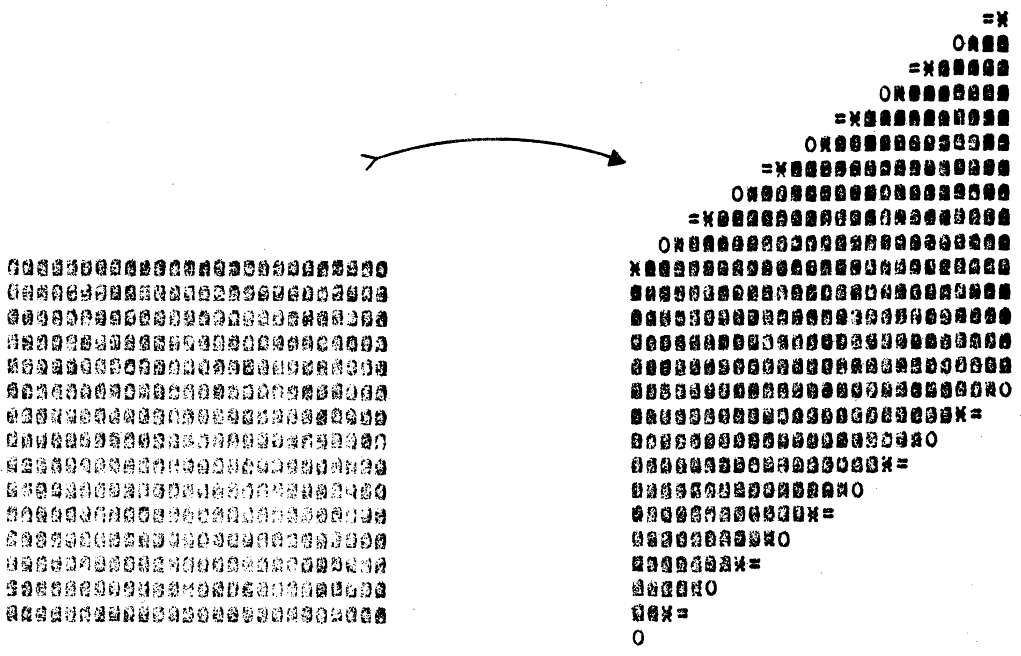


Figure 5b: Illustration of Digitized Shearing

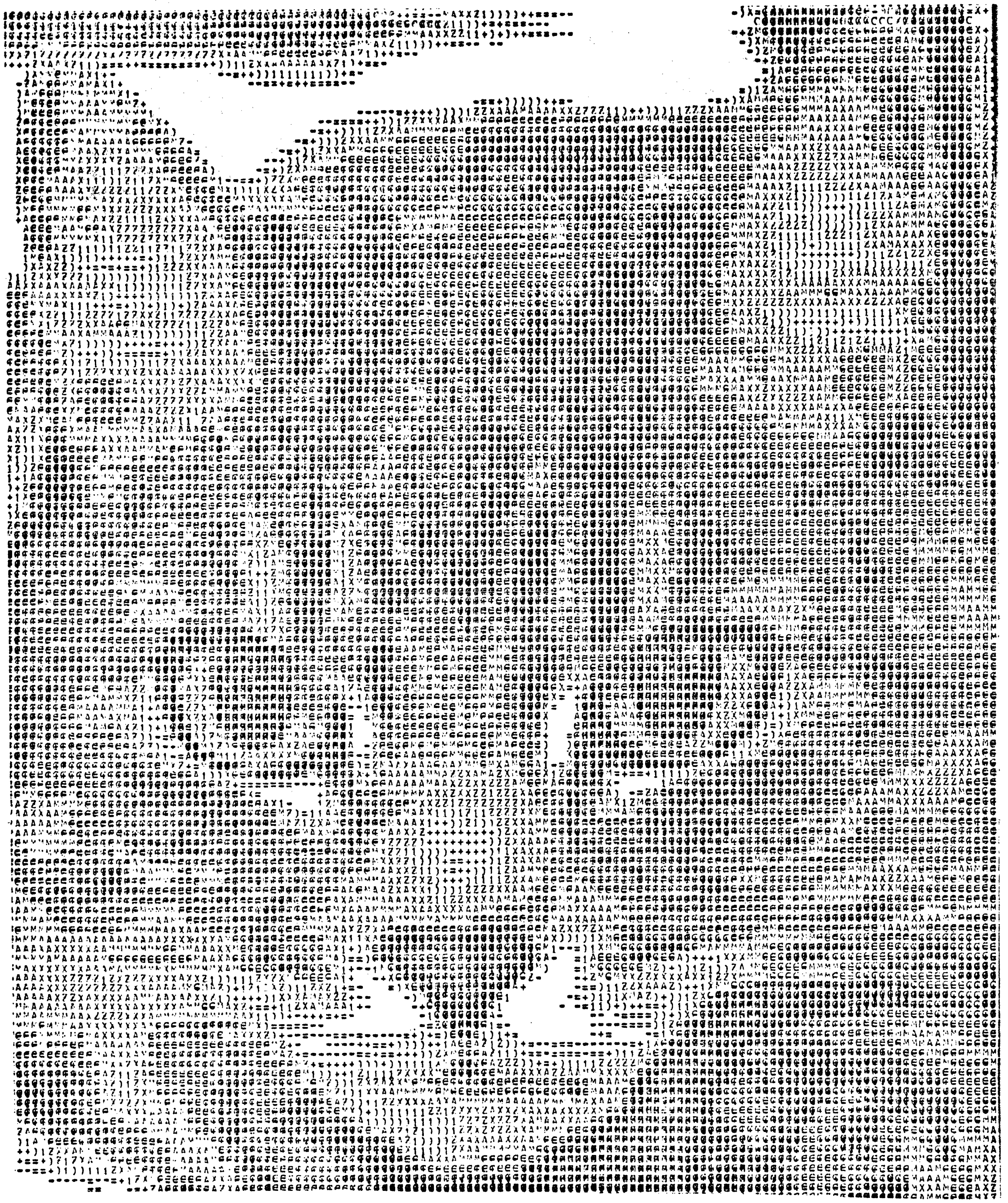


Figure 6: Digitized Cat

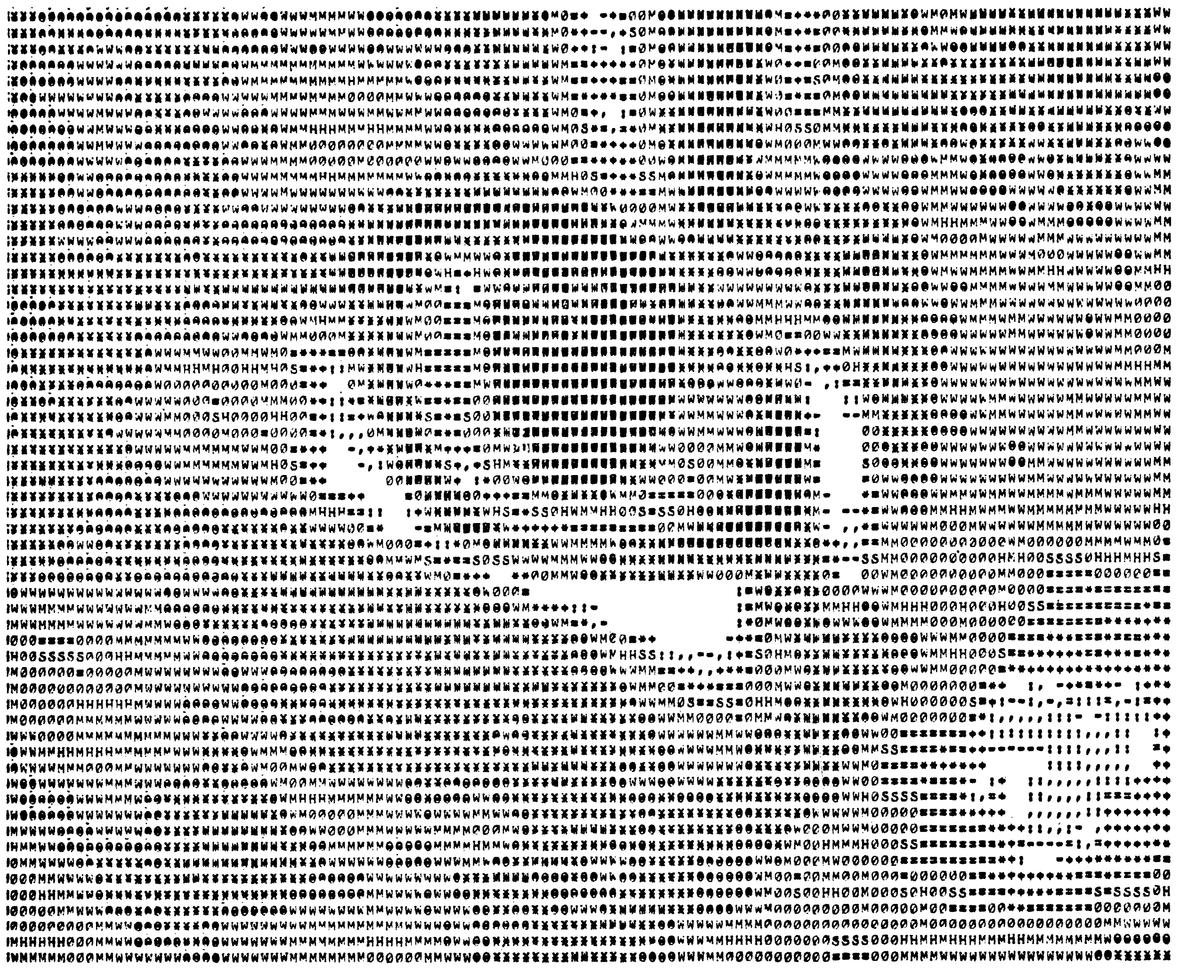


Figure 7: Digitally Magnified Cat's Eye

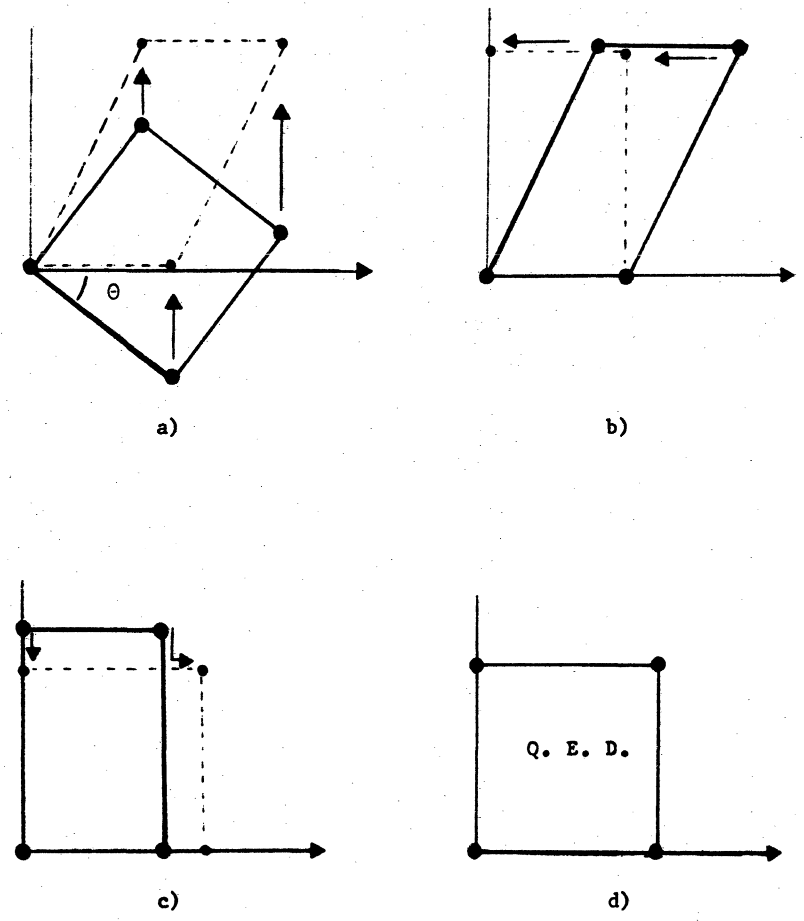


Figure 8: Decomposition of Rotation into Axis-Parallel Shearing and Scaling

ENHANCEMENT OF COMPUTING POWER IN MULTIPROCESSOR SYSTEMS FOR PROCESSING OF DIGITIZED PICTURES

K. Vorgrimler
Forschungsgesellschaft für Angewandte Naturwissenschaften e.V. (FGAN)
Forschungsinstitut für Informationsverarbeitung und Mustererkennung (FIM)
7500 Karlsruhe 1, Breslauerstr. 48
W. Germany

Abstract -- There is an increasing trend to solve picture-processing tasks on computers. The computation of local homogeneous window operations (convolution) tends to be impractical when a conventional computer is used because of the resulting time requirements. A structurally programmable multiprocessor is able to solve these tasks in one to two orders of magnitude faster. The principle of operation of the system and the individual processor are presented.

Introduction

Pictures serve as the primary information in a great variety of fields of interest for scientific research. Examples are the area of both bio- and/or human medicine with cellular analysis or evaluation of X-ray-pictures. Another important field of application of picture processing techniques is connected with various satellite programs, where weather forecast, surveillance of industrial emission or vegetation etc. are the aims of picture interpretation.

Usually picture processing is dividable into five functionally different steps:

- picture scanning and digitizing
- picture preprocessing
- feature extraction
- feature analysis
- classification.

One major problem in this processing chain is the almost unrealistic computing time or necessary computing power in the domain of picture preprocessing. Using a conventional uni-processor to perform these tasks results in computing times of a few minutes to several hours depending on the algorithm and picture size.

With a special multiprocessor configuration actually being constructed at FIM, the implementation of many preprocessing algorithms proves to be faster by one to two orders of magnitude compared with the use of a conventional computer for the same task.

Local picture processing

Generally there are two mathematical techniques available for the implementation of picture preprocessing principles like spatial filtering. In a computer a picture is usually represented by a two-dimensional point-matrix of grey-levels. One technique - convolution - is applied directly in this grey-level-domain whereas the second technique is applied in the so-called frequency-domain. This domain is obtained after the application of an integral transform e.g. Fouriertransform to both the grey-level picture and the respective filter. The philosophy of this technique is that relatively complex operations like correlation in the frequency domain can be obtained by simple matrix multiplications. This technique has attained a practicable aspect after the presentation of the Fast Fourier Transform algorithm by Cooley and Tukey.

In local picture processing a special application dependent evaluation matrix is applied to a picture area of corresponding size (fig. 1).

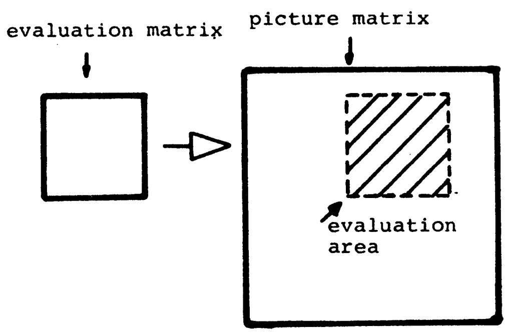


fig. 1: Local picture processing

Fourier techniques are used in such cases where the evaluation matrix covers a great picture area or the entire picture. But in contrast to the advantages of this technique there are mainly two grave limitations:

- In addition to the fast matrix multiplication three time-consuming transformations are necessary: picture transformation, filter transformation and the transformation of the product back to the original domain.
- Due to the linearity of the transform the use is restricted to linear operations whereas in many picture preprocessing applications nonlinear logical operations or thresholding are highly efficient.

In the grey-level domain most operations are so-called window operations. These simple local operations have the following properties:

- The window (evaluation area) implies a relative small neighbourhood e.g. a square picture-submatrix of 3x3 up to 11x11 picture elements.

- Window operations are position-invariant or homogeneous which means that the evaluation function remains unchanged when the window is shifted point by point over the entire picture.

As a simple example an algorithm known as "stroke difference" is presented which leads to a "derivative" of a given picture B. With the submatrix-notation in fig. 2 the stroke difference is given by

$$\Delta B_{pq} = \frac{1}{2}\left[\Delta B_{pq_x} + \Delta B_{pq_y}\right]$$

where (1)

$$\Delta B_{pq_x} = \frac{1}{3}\left|(b_{-1,-1}+b_{-1,0}+b_{-1,1})-(b_{1,-1}+b_{1,0}+b_{1,1})\right|$$

$$\Delta B_{pq_y} = \frac{1}{3}\left|(b_{-1,-1}+b_{0,-1}+b_{1,-1})-(b_{-1,1}+b_{0,1}+b_{1,1})\right|$$

As depicted in fig. 2 the result is related to a position within the resulting-matrix which corresponds to the position of the central element of the window in the original picture-matrix. The edge-elements in the resulting-matrix in fig. 3b are supposed to be filled with zeros.

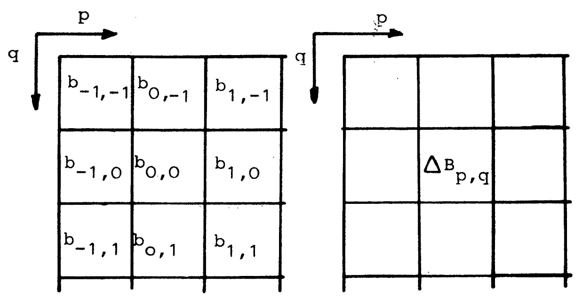


fig. 2: Notation within a 3x3 window

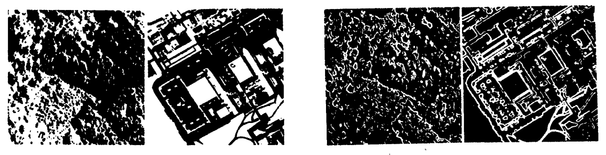


fig. 3: Stroke-difference applied on picture B

Fig. 3a shows an aereal photo and fig. 3b the result after the evaluation of the stroke-difference algorithm. The resulting values of the stroke difference are displayed as grey levels.

Cascading of window operations

To exploit a given analytical expression with regard to the degree of its inherent parallelism the simplest way is to depict the corresponding computing graph. For the simple expression given in (1) this is shown in fig. 4.

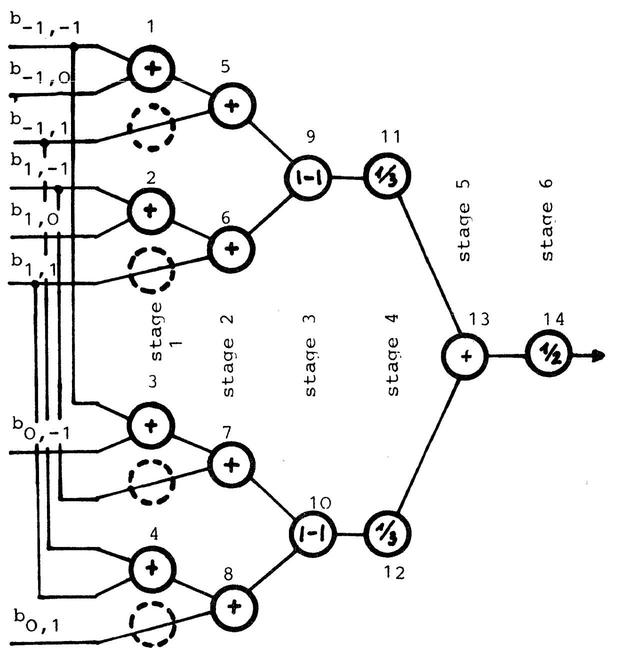


fig. 4: Cascading of stroke difference

The evident parallelism in a computing graph usually is not exploited in practice. The user of a conventional computer streches the parallelism into a task suitable for a one-at-a-time hardware. Assuming for simplicity that each of the operations within the circles in fig. 4 require one time-unit of occupation in an uni-processor-equipment, a space-time diagram results as depicted in fig. 5.

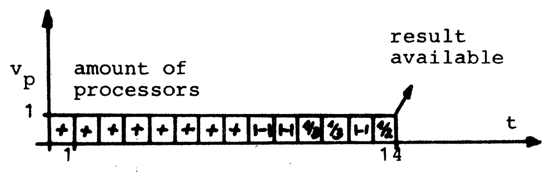


fig. 5: Space-time diagram for one processor

On the other hand the minimum processing-time is obtained when the problem-inherent parallelism could be covered by an appropriate hardware multiplicity (fig. 6).

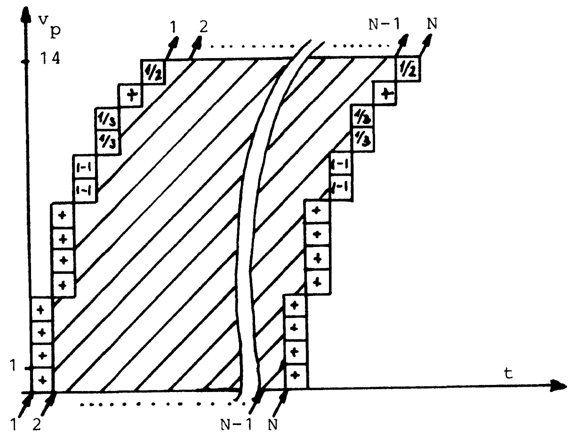


fig. 6: Complete coverage of parallelism

Note that the coverage in the sense of fig. 6 requires to redefine the conception of parallelism. In array computers a number of processors with identical properties work on a set of multiple data. In the complementary pipeline computer a single data stream is submitted to a sequence of operations within the processors forming the pipeline. In this case the processors work simultaneously on the single data stream where data coexist within the pipeline at a different processing state. By this way a kind of pseudo-parallel processing - better simultaneous processing - takes place. When data are available on the respective data buses (fig. 4) processors 1 to 4 can start simultaneously with the addition of their input-data. In the space-time diagram (fig. 6) this fact is represented by the occupance of 4 units in the first time-interval. When the addition is completed these processors can transmit their results to the units 5,6,7 and 8 respectively etc. Note that the final result is available after a "filling time" of 6 time-units corresponding to the 6 stages in the cascade.

The next input data can be offered to the system after the first time interval when the four "pseudo-processors" represented by dotted circles (fig. 4) are inserted. These processors are simple buffer-registers capable of holding the data during one time interval to avoid conflicts. Note that without the buffers their respective input data could not be changed until stage 2 would have completed the operation. Now the first processors can start operating on the second data-set when the following stages are still working on the first etc. This results in the same effect as in linear pipelines, namely that results are available in the same rate as input data are supplied.

The configuration shown in fig. 4 could be realized and fix-wired for the given algorithm. Beyond that provisions must be made to control the data flow and the synchronisation of the different processing units. Under the aspect of flexibility this solution would be a grave restriction and would lead to an immense number of special circuitry. Although this method cannot be excluded for a set of frequently used operations, a more flexible system requires a programmable structure. This means that the individual processor-properties and their mutual interconnection must be controllable by a program.

The Flexible Multi-Pipeline-Processor-System (FMPP)

The system consists of a set of (max. 64) originally isolated processors and a set of data connections (B_0 ... B_{61}). Four of these processors are shown in fig. 7. At the output each processor is associated to a single of these data paths as indicated by dotted lines.

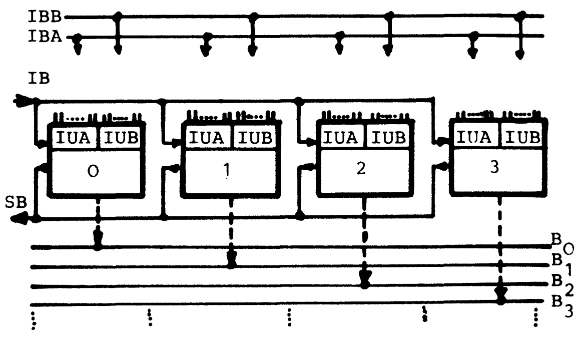


fig. 7: 4-Processor-System (non-programmed)

Additionally two data buses (IBA, IBB) are available. The buses IB and SB are common to all processors. IB serves to transmit the instructions to the individual processors, SB retransmits selectable internal processor status. The buses can uniquely be used to transport information in the directions indicated by arrows.

To establish a desired configuration the processors are sequentially programed via IB and the transferred instructions are stored within each processor. Each processor is realized as a three-address machine, this means that one single instruction contains two operand-source addresses and one operand destination address in addition to the operation code. Each processor has two independent input-control-units (IUA, IUB) affected by the destination parts of the instruction code. Each of these units is capable of establishing a connection to one of the existing data buses (e.g. 64) at a time. As the two input-units are independent, they can fetch two operands from two different buses simultaneously. The output is controllable by an output-mode-control code which allows results to be transfered directly via the corresponding bus or to be served in a processor-internal register stack for further use.

After the proper programming a configuration shown in fig. 8 can be "switched" together. It should be remarked that the interconnections shown are not necessarily everlasting and consisting simultaneously. The data are buffered at the end of the sending equipment (processor) and the transfer only takes place when the receiving unit (processor) requests data exactly on the respective bus.

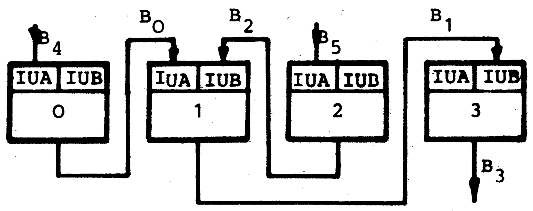


fig. 8: Processor interconnection after appropriate programming

Note that the buses IBA, IBB, SB and IB are omitted for simplicity. The interconnection shown could be used to implement the last three stages in the cascade in fig. 4 when the input-data are submitted via B_4 and B_5.

It should be pointed out that the two additional data buses IBA, IBB are organized as selector buses. This was made as a concession to the preliminary use of the system and will be explained later.

Basic building blocks of each processor

Each of the processors within the multiprocessor-system includes the subunits shown in fig. 7. Two input-units (IUA, IUB) control the transfer of data into the attached register stacks (RSA, RSB) or directly into the arithmetic logic-unit (ALU) respectively. The output (OUT) unit controls the transport of intermediate results via an output data-bus. Output data can be selected directly from the ALU or from a register-stack (RSC). Each processor has a bipolar instruction memory (IM) capable of holding up to 256 instructions. Attached to this memory is a decode and control unit (DECC) and an instruction-input unit (IIU) which control the loading of instructions into the memory via the instruction-bus (IB). A status-bus (SB) with a corresponding control unit (STU) serves to transmit some selectable internal conditions and is used for tests and processing control. It should be pointed out, that all connections for data or instructions from or to the processors environment are physically existent. The buses IBA, IBB, IB, SB are organized as selector-buses. This means that all processors within the system are connected in parallel to these buses and a transfer over them must be established logically. The data buses (B_0-B_{63},IBA,IBB) at the inputs are data paths of the system-internal multi-bus-system. They are organized to handle the input or output of one single word from or to the environment, depending on the respective input- or output-instruction.

As the use is mainly restricted to picture processing tasks, the word-length is adapted to those requirements. All data buses are 8 bit wide with a set of supplementary control lines. The word-length of a single instruction is 32 bits subdivided in 4 bytes. The first byte controls the function of the ALU, whereas the next three bytes are the addresses of the selected input and output data sources respectively.

The technology used is standard and low Power Schottky-TTL, requiring a total of 200 packages per one processor.

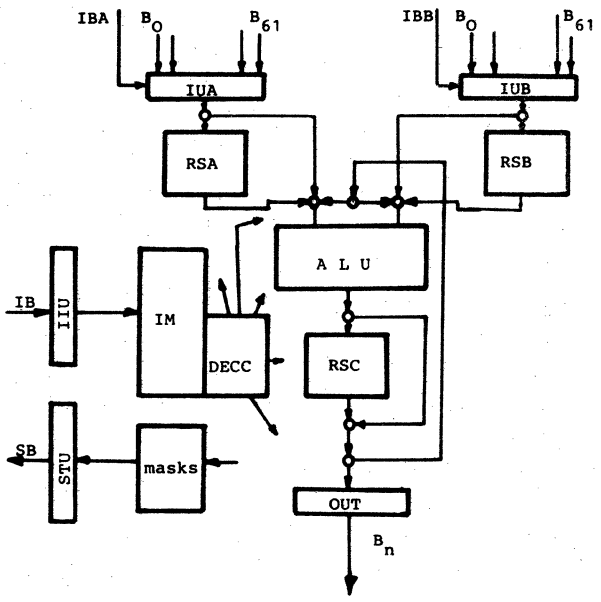


fig. 9: Building blocks of processor n

Principle of operation

To explain the asynchroneous and data-controlled principle of operation a simplified evaluation-net-representation of a single processor will be used as depicted in fig. 8. Note that this representation here is used only as an informed descriptive method. For details the reader is referred to [1].

The actions within a processor can be described by a set of transitions (depicted as horizontal lines). Connected to these transitions there are a set of locations (circles) and resolution locations (hexagons).

Both data inputs of the processor are depicted as N-way-input-switches. The resolution locations r_A and r_B hold the address of the data path to be selected. The transition fires if the selected input location (data bus) contains a valid data and location (b_A, b_B) is empty. After the transition, data are removed from the corresponding input locations and placed on the output locations (b_A,b_B). Depending on the values of the resolution locations r_{A1} and r_{A2} and the contents of the associated locations, the firing of the transitions a_1 and a_2 may be activated and hence the filling of b_{A1} and/or b_{A2}.

These locations are the inputs for the processing-transitions a_p carrying out the data alteration placed in b_C. Transition a_C controls the way of the result either to locations internal to the processor (b_{IC}) or to its environment (B_n). It should be noted that in the formal abstraction of the processors activity given in fig. 8 one single instruction of the processor contains information concerning the following:

- value of r_A, r_B, r_{A1}, r_{A2}, r_C
- content of b_{IA}, b_{IB}
- transition procedure of transition a_p

Depending on this information (instruction) the processors activity is either controlled by the content of the peripheral locations B_O-B_{63},IBA,IBB or the inner locations b_{IA}, b_{IB}. Note, that the instruction is completed after the firing of transition a_C and a new instruction is fetched.

By this pipelining of transitions the processed activity is triggerable by the presence of valid data at the peripheral locations. At the input side the activity is interrupted until the firing conditions of a_A a_B are fulfilled (content of selected location). At the output side the activity stops when the data at B_n has not been removed.

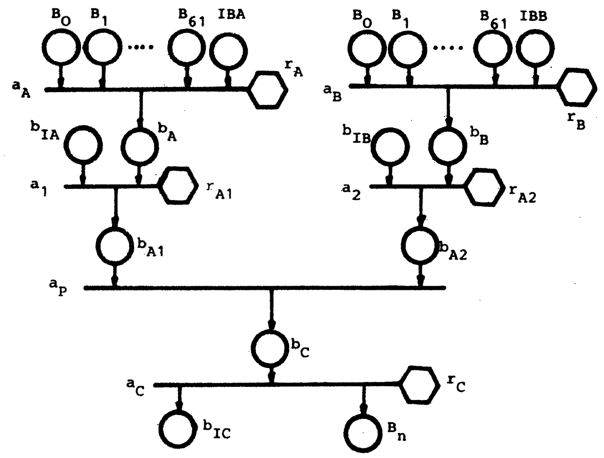


fig. 10: E-net representation of a single processor n

By this way the programs within the distributed processors contain not only their individual processing properties (a_p) but also the sources and destination of data. In addition to that no superior timing control mechanism must be provided to synchronize the active processors since data flow itself acts as start-stop-signal for the individual unit. Furthermore no internal clock is needed because each instruction consists of a chain of transitions with their individual firing conditions and time requirements and the new instruction is only initiated after the firing of a_C.

To compute a given window operation the user has to subdivide the entire task into a cascade as depicted in fig.4 with the notational understanding that there are a limited number of processors available. As a single processor is capable of holding up to 256 instructions this limitation is not grave. As in a linear pipeline the traversal time of a single task is mainly determined by the "slowest" pipeline segment. This fact also holds true in a "mixed" configuration, because due to the data dependent control no critical races can occur. The structure adapts itself to the slowest segment. Therefore, as a general rule, the task should be divided in a number of subtasks, each as small as possible (small number of instructions). As pipelining and parallel processing is combinable there is a high degree of freedom to handle the trade-off between the length of the subprograms and the number of processors to be used. Once the structure is fixed the user transforms it to the adequate processor configuration by programming each of the processors. The programs are delivered to the processors via the instruction bus (IB) in fig. 7. After the programming phase the structure is fixed and additionally a mechanism must be provided to deliver the input data to the configuration as well as to transfer the results back to the picture storage.

Realized system configuration

Fig. 9 shows the preliminary location of the FMPP as a peripheral equipment of a PDP 11/45 minicomputer.

In addition to the FMPP, 3 supporting modules are necessary. The input-output-interface (I/O Int.) delivers the programs to the submits and to the single processors within the FMPP via the instruction bus (IB) during the programming phase. After programming the properties of the system are fixed and processing is taking place according to the data-rate of picture-data transferred to and back from the system. The output-interface handles the transfer of the results back to the PDP's memory which acts as a buffer for the picture data normally stored on disks. The I/O-interface also serves to transfer selected status information from the processors back to the host computer.

In this mode of operation two difficulties arise:

- picture data can only be transferred serially via the UNIBUS so that the multiple stream of input data for the FMPP is not available di-

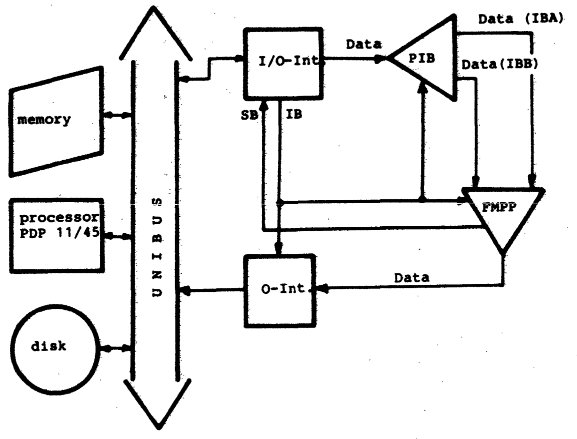


fig. 11: Part of picture processing system

rectly;

- as the window is shifted point by point over the picture, each pixel belongs to n^2 different locations of the window, n being the dimension of the used submatrix. This would require to transport roughly $(nxW)^2$ pixels (N= dimension of picture) to the peripheral equipment;

A multiport semiconductor-memory actually being constructed at FIM is expected to be operational in late 1977. In order to get familiarized with the system the decision was made to put it to work preliminarily in the environment shown in fig. 9 by adding a third module. The programmable input-buffer (PIB) spreads the single data stream. It essentially consists of a set of interconnectable (program controlled) shift registers. Each register row has a capacity to store 1024 pixels.

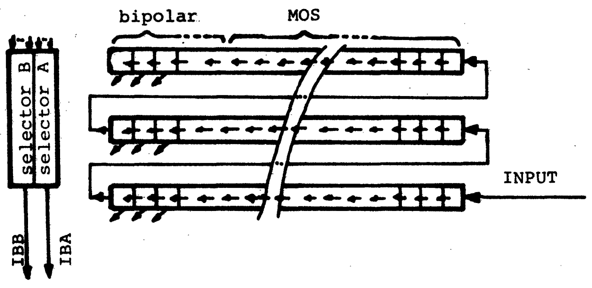


fig. 12: Operation principle of the PIB

Each row consists of static shift registers realized in MOS-technology and a supplement of 16 registers in TTL-technology. These "tail"-registers are random accessable. In fig. 12 the interconnection of 3 rows is shown. This interconnection is chosen when an operation requires a 3x3 window. Instead of shifting the window over the picture, within the PIB the operation is inverted by shifting picture data below the fixed window. While the data input-rate is relatively slow due to the access time of the mass-storage (disk) the output data (window) can be transferred to the FMPP at high speed. This is done via the two previously mentioned selector-buses IBA, IBB according to a delivering-program stored within the PIB. The two buses have their own program-memory and transfer control units so that they can operate simultaneously. With the use of the PIB it is only necessary to transport picture data once to the peripheral equipment. When the shift registers within PIB are filled with 3 picture-rows (fig. 12) the operation can begin. The first result must be transported back via the output-interface, then a new input pixel activates the computation of the next result etc.

The transport of a picture with the size of 1024x 1024 to and back from the peripheral system requires roughly 10 seconds. This time results when the specifications of all the building blocks in fig. 11 are taken into consideration. The realized system is configurated so that all the window operations presently used or having been developped at FIM [2][3] can run in this time, when applied on a picture of the size mentioned. This fact corresponds to a speed increase by a factor of 7 up to 300 when it is compared with the run times for the computation of the same algorithms on a CDC 3300 computer. The speed increase depends on the amount of parallelism of the given algorithm and the number of picture elements required within the submatrix for its computation. Note that some non-linear operations like the stroke difference (1) do not use all pixels within the window.

Conclusion

Hardware parallelism and pipelining are combinable to cover a maximum of parallelism inherent to a given algorithm. Due to the data controlled mode of operation the desired structure can be established by programming the individual processors without the need of a special equipment to control the mutual data interconnections. The data flow in a cascade is unidirectional, so that the control units are relatively simple justifying the additional implementations of these units in each of the processors.

By adequate programming the FMPP can serve to overcome one of major problems in picture preprocessing, namely the unrealistic processing times necessary when conventional computers are used. From the user's point of view a great disadvantage is the fact, that for the moment almost all software support is missing, so that the programming is rather cumbersome. The future work will overcome this problem by developing software basing on a simple assembler at the single processors level. It should finally be noted that the software support must be provided by a general purpose host computer. Presently 3 processors have been realized. The system with a preliminary number of 16 individual processors is expected to be operational in early 1977.

REFERENCES

[1] G.J. Nutt "The formulation and application of evaluation nets"
Comput.Sci.Group, Univ. Washington, Seattle, TR 72-07-02 1972

[2] F. Holdermann, H. Kazmierczak "Processing of Gray Scale Pictures"
Computer Graphics and Image Processing, Vol 1, No 1, 1972

[3] K. Vorgrimler "Zur Leistungssteigerung von Mehrprozessorsystemen für die Verarbeitung digitaler Bildinformation"
Dissertation Fak. Elektrotechnik, Univ. Karlsruhe, 1976 (being printed presently)

APPLICATION OF DISTRIBUTED PROCESSING TO THE PRODUCTION OF DIGITAL TERRAIN DATA*

Dennis E. Moellman
Defense Mapping Agency
Washington, D. C.

Robert A. Meyer
Department of Electrical
and Computer Engineering
Clarkson College of Technology
Potsdam, New York

Abstract -- A distributed computer network is described which forms an integrated system for the production of digital terrain data from stereo aerial photography. This system includes on-line processing of data collected by high-speed digitizing instruments, man-machine interactive editing capability, and a centralized processor for managing inter-processor data transfers. This paper analyzes the system requirements in terms of specific architectural features which must be provided. We describe the use of a SIMSCRIPT simulation to test the feasibility of the basic design concept. Simulation results were also used in determining design parameters such as the number of processors, memory size, and expected throughput rates. Significant characteristics of the system such as modularity and reliability are discussed.

I. INTRODUCTION

In a recent paper [1] concerned with computer interconnection structures, Anderson and Jensen discuss the lack of published material describing the basis for design of these systems or making a comparative evaluation. This paper takes a step toward filling that need by presenting the design of a distributed computer network together with an analysis of the system requirements which led to this specific design. Our goal is not to describe the implementation details, but rather to provide the reader with a view of the design process and an understanding of the relationship between system requirements and the network architecture.

The distributed computer network (DCN) has been designed to solve a real-time system integration and data processing problem encountered in the production of digital terrain information by the Defense Mapping Agency (DMA). In a mapping sense, an aerial photograph represents a state-of-the-art high-density storage medium for storing terrain information. Two such photographs appropriately exposed comprise a stereomodel of the earth's surface from which three-dimensional terrain data can be extracted. The most efficient means for extracting such data is to digitize the analog information contained within each photograph so that the photographic density (gray shade) of each picture element (pixel) is represented by an integer. Sophisticated correlation algorithms coupled with perspective geometry calculations are employed to determine the three-dimensional relationship of a specific point to a reference datum. A collection of such points covering a particular area form a product called a digital terrain data base.

In order to meet an ever increasing demand for digital terrain data bases, DMA has sought to increase their productivity by utilizing digitizing instruments which operate automatically with little or no manual intervention. These newer digitizers are capable of operating between 10 and 40 times faster than the previously used manual instruments. The immediate problem resulting from such a change is the inability of the existing off-line, batch computer system to meet the increased processing load. The problem is further complicated by the additional requirements for manually prepared initiallization data and limited manual editing capability.

A block diagram of the DCN is shown in Figure 1. This system was proposed by the group at Clarkson College and has been accepted for implementation by DMA. At the present time detailed software specifications for the system are being written at Clarkson. The final design resulted from the contributions of several people at Clarkson, DMA and the Rome Air Development Center (RADC). In the remaining sections of this paper, we describe the operating environment, analyze the system requirements, and discuss the important characteristics of the design including the application of system modeling and simulation techniques to predict expected performance.

II. BACKGROUND

Digital terrain data bases are rapidly replacing conventional line maps. This is particularly true because advancements in computer technology have influenced navigational systems so that the application of digital terrain data is more economical and practical. Digital terrain data can now provide radar images for flight simulation and navigation by real time on-board correlation as well as terrain profiles between two points for flight planning and conventional contour maps [2].

Recognizing that present equipment could not effectively satisfy the increasing demand for digital terrain data, DMA contracted with Bendix Research Laboratories for the development, under the direction of RADC, of a new digitizing device, the Automatic Compilation Equipment (ACE). The prototype instrument consists of a conventional manual unit retrofitted with a laser scanner and a digital correlator consisting of two microprogrammable minicomputers. Based on new concepts in scanning and digitizing, the ACE is able to scan and digitize each pixel of a stereo

*This work was supported by the Defense Mapping Agency Aerospace Center through Rome Air Development Center and the Post Doctoral Program.

pair of photographs, perform automatic digital correlation (image matching), and compute 58 terrain elevation profiles simultaneously. The collection rate is 250,000 points in approximately 10 minutes at a density of 6500 points/square inch [2].

In order to achieve these rates an ACE must operate in an automatic mode which produces raw data in the form of irregularly spaced terrain profiles in a local coordinate system. The raw data must then be transformed to global geographic coordinates and interpolated to a uniform grid of elevation values.

Two difficulties with completely automatic digitizing are the inability of the machine to exactly track terrain peaks and valleys and the loss of correlation in adverse areas of the photograph. These problems could be solved by relying on partial manual operation; however, this would seriously degrade the overall efficiency of the ACE. An alternative formulated by DMA is the use of currently available, manual digitizing instruments to produce additional data for each stereopair of photographs. This additional data includes peaks and valleys, "fill-in" areas not digitized by an ACE due to poor correlation, and also certain information used by an ACE operator to reduce the setup time prior to ACE operation. Several of these manual digitizing instruments have been linked together with a host minicomputer to form the pooled minicomputer system [3].

The ACE's and pooled minicomputer system may therefore be viewed as sources of raw input data which must be processed and edited in order to be acceptable to the user. The rate at which large volumes of data are being collected clearly indicates that off-line data transfers (such as magnetic tapes) and processing must be replaced with an integrated on-line system.

III. SYSTEM REQUIREMENTS

The operating environment described in the previous section provides a basis for determining the computational requirements of the proposed system. These requirements may be divided into five major functional tasks:

1) data collection
2) processing
3) editing
4) file management
5) job control.

Each of these tasks places specific demands on the system and thereby influences the overall architectural configuration. In this section, we analyze these requirements in terms of a general processing system and show how the DCN meets these demands.

Data collection consists of accepting input data from two distinct categories of sources: real time and non-real time. Real time input is received from each of two ACE's at the rate of 1600 words/second (16-bit words) per ACE. These transfers occur as 192-word blocks which must be received every 120 milliseconds. Failure of the collecting processor to perform the transfer within the stated time period results in loss of data.

The second category of input data is produced by the pooled minicomputer system. This system contains sufficient local storage that a data transfer may be deferred until the collecting processor requests it. It should, therefore, be possible to use a single processor to receive input from all collection devices. This processor is called the input processor in the DCN (see Figure 1). During periods when one or both ACE's are not actively operating, the input processor requests the current backlog of input data from the pooled minicomputer system.

The processing task includes two basic operations which must be performed on the data. As described in Section II, the ACE output consists of terrain data samples giving a location and elevation, (x,y,z), in the local coordinate system of the stereophotographs. The first operation required is a coordinate transformation which maps the triple (x,y,z) into a geographic coordinate system. The transformation requires 22 multiplications and 20 additions using 32-bit floating point arithmetic and approximately 13 additional load and store instructions. Since the transformation is applied to every input data sample from an ACE, the system must be capable of executing about 3200 floating point instructions for each 192-word block of 58 samples. In practice the system processing capability must be somewhat higher to allow for the overhead associated with the I/O operations. Of course, the transformation could be done in non-real time. However, in the next paragraph we present compelling reasons for providing sufficient speed to perform the transformation in real time.

The second operation to be performed is interpolation. Unlike coordinate transformation, the interpolation function requires all of the input samples in a neighborhood of the output point. Therefore, the data should be sorted prior to interpolation so that points within a neighborhood may be easily located. Since a pair of stereophotographs produces approximately 5.0×10^6 words of data, a conventional batch sorting procedure could be very costly. The solution we have proposed sorts the data into tractable geographic regions as it is being collected. Thus, the primary data structure for interim storage allows one to access a particular region of any geographic point within the domain of the stereophotography. Since the sort is performed on the basis of the geographic coordinates of a sample, it is necessary to execute the coordinate transformation as the data is collected and before the interim storage structure is built.

The interpolation algorithm requires approximately 16N floating point operations per output point, where N is the number of input samples in the neighborhood of the output point. In a typical situation a pair of stereophotographs will cover 7.5×10^5 output points with N = 8. The expected maximum operating rate of an ACE is one stereopair per hour giving a computational requirement for interpolation of 2.7×10^4 floating point operations per second for each ACE.

To summarize the processing load on the system we find that each ACE demands the execution of 5.4×10^4 floating point operations per second. These calculations do not include time required for sorting or I/O. In order to support at least two ACE's (and perhaps three in the future) with a reasonable safety margin in the timing, we partitioned the processing task in a natural way into two parallel operations, coordinate transformation and interpolation. As described previously coordinate transformation should be performed in real time as the data is collected and sorted; this operation is done by the input processor. A second processing capability is provided for interpolation. The interpolation processor could be either a single processor with an average floating point instruction time of less than 10 μsec. or two processors with appropriately slower hardware. The primary decision criterion is cost.

Editing of the input data is required after collection to check validity and to identify areas which require manual fill-in. Editing is also required after interpolation to insure overall consistency. Given the volume of data which must be examined it is necessary to automate as much of the editing task as possible. As data is collected it can be separated into two groups, good quality and poor quality. This separation is made on the basis of a correlation coefficient associated with each input sample and can be performed automatically. Additional editing requires manual intervention and is performed with a man-machine interactive graphics facility consisting of several minicomputer controlled CRT displays. To service this facility requires data format conversions and selected subfile retrieval and update operations. These are performed in the DCN by the edit processor.

The most complex problem to be solved is data storage and management. A single typical file will consist of approximately $5. \times 10^6$ words. At any time, we expect about fifteen files to be active thus requiring 7.5×10^7 words of readily accessible storage. Read/write requests may be generated by the collection, editing and interpolation tasks. Since these are concurrent operations, a means for coordinating simultaneous requests must be provided. A single file processor provides this capability in the DCN. Active files are stored on two 42M word disks with magnetic tape backup.

A second aspect of the data management problem concerns the organization of individual files. Although the choice of data structure has not directly influenced the hardware features of the network architecture, it has been a consideration in the design of the software message handling system. Briefly, each file is composed of a two-level hierarchy of subfiles. The first level partitions the data by collection source and the second level by geographic region. Since a subfile is of variable length to allow for data addition or deletion, a convenient data structure is a linked list. Each item in the list is a block of 128 words corresponding to one disk sector. The bulk of messages between processors is data for either file storage or retrieval, and therefore a message consists of a variable number of 128-word blocks.

Job control is a system level task designed to meet not only the processing requirements of the operating environment, but also the needs of the computational system itself. For the DCN we have a collection of nearly autonomous processors, each performing a specific task. The goal of job control is to insure that a set of input data is processed according to a prescribed procedure and that efficient use of system resources is made. Since the file processor is the only centralized processor with access to all data, the job control task is executed in the file processor.

IV. SYSTEM CHARACTERISTICS

Within the context of [1], the DCN is a hybrid network. It most closely resembles the "star" architecture which is defined as a set of processors, indirectly connected through a centralized routing mechanism using dedicated message paths in a star shaped arrangement. In the DCN the central switch is a processor itself, and thus we have both direct and indirect processor-processor interconnection. However, from the job processing viewpoint, the file processor is transparent and serves only to direct the data flow. The analysis of system characteristics we give in this section will demonstrate the similarity of the DCN to a "star" architecture in terms of advantages and disadvantages.

Perhaps the most commonly used term in describing distributed computer systems is modularity. Except for the file processor, the DCN is clearly modular; that is, the addition of another processor requires only another link into the file processor. If the system grows to the point that the file processor is overloaded in terms of computational power or I/O ports, then the system architecture can be preserved only by replacement of the file processor with a higher performance machine. Therefore, an important consideration in the design of the DCN was to allow for future expansion in determining the performance specifications of the file processor.

A second characteristic which is closely related to the hardware modularity described above is software modularity. If a new processing step is added, the only change required is the modification of a job flow table in the file processor.

The new step may be implemented on any of the processors with sufficient computational power available.

Flexibility refers to the ability of a system to meet changing demands placed on it by the operating environment. Clearly this is similar to the concept of modularity, but we restrict the notion of flexibility to short term adaptation as opposed to long term system growth. A key feature of the DCN which contributes to its flexibility is the homogeneity of the processors. The only difference among the processors are memory size and I/O configuration. Thus, it is possible to reallocate certain operations among the processors as a means of relieving temporary bottlenecks in the overall job flow.

The concept of flexibility is also important in responding to a failure within the system. The critical element is clearly the file processor since loss of this element blocks access to the data base. This is essentially the price one must pay for a centralized access to the data base. When a failure occurs which can be corrected within a few hours, each processor continues to operate until interaction with the network is required and then it waits for restoration of service. In the event of longer term failures, the file processor can be physically replaced with another one of the processors. This is a form of graceful degradation since it would require suspension, or at least substantial reduction, of the tasks previously performed by the replacement processor. Thus, the flexibility of the DCN contributes to total system reliability.

In analyzing the overall characteristics of the DCN one should examine the cost/performance ratio as compared with alternatives. A detailed comparison of this sort is beyond the scope of this paper, but we will summarize an analysis presented in the hardware specification prepared for DMA.

Three major alternatives were considered: 1) a single large scale general purpose processor operating in a real time foreground/batch background mode; 2) a dual processor network with a shared data base using dual port disks; 3) a four processor network interconnected with a set of switches such that any single processor failure can be tolerated.

In comparison to the DCN, the first alternative is considerably more expensive, less modular, and less reliable. Although a single processor may offer the opportunity for greater flexibility, this would probably be achieved at a higher cost for more complex software.

The second alternative does not have the restricted modularity and reliability associated with the file processor in the DCN. In this case the major difficulty is data management with a decentralized control for accessing the data base. Lack of coordination among disk I/O requests poses problems with respect to data integrity, file maintenance (e.g. garbage collection), and insuring the efficient flow of data from one job step to the next. We estimate this alternative costs approximately the same as the DCN and provides a lower overall performance.

The reliability of the third alternative would appear to be significantly greater than for the DCN. A careful analysis reveals that while the critical element is no longer the file processor, the processor interconnection switches are now critical. Thus, in this case, system reliability depends on a set of hardware switches which are probably not off-the-shelf items as compared with a single processor in the DCN. We believe the DCN is a better choice. Modularity and flexibility are better for the third alternative but at a greater hardware cost and system complexity.

V. SYSTEM SIMULATION

In designing a large, complex computer system such as the DCN it is important to verify the feasibility of the basic design concept. Given this particular design, one must then estimate the necessary processor specifications such as memory size, instruction execution time, and interrupt response time. A useful tool for solving these problems is simulation. Our approach to system simulation and the kind of information it can provide are described in this section.

A discrete event simulation language, SIMSCRIPT II.5, was used to simulate the DCN. The DCN is modeled as a set of tasks to be performed by the processors where each instance of a specific task is an event. Events may be scheduled externally by the user or internally by the system. A set of task queues is provided in each processor to hold pending requests for processor service. For example, the file processor maintains a queue for disk I/O requests. When the processor wishes to access the disk, it places a request in the disk queue and schedules an I/O taks. If the disk is unavailable, the request remains in the queue. When the current disk operation is scheduled to end, the next request in the queue is serviced. The level of detail included in the simulation provides for modeling the disk in terms of rotational speed and head movement from cylinder to cylinder.

The statistics gathered during a simulated operating period for each queue are:

1) number of requests waiting, average and maximum;

2) total length of requests, in words, average and maximum;

3) waiting time per request, average and maximum;

4) total number of requests serviced.

Additional statistics onthe disk search/read/write times are also maintained. Feasibility of the design concept is verified by observing that the system operates with an acceptable throughput based on reasonable estimates of processor speed. The statistics for queue lengths are indicative of the required memory size in each processor. Changing estimated processing time allows one to determine lower bounds on processor speed.

Simulation is also a useful tool for performing a sensitivity analysis on the system. Since many of the processing times used in the model are only estimates, the sensitivity of the system to these quantities should be determined. Recalling the discussion of system modularity, the critical element for sensitivity analysis is the file processor. The results of our studies indicate that the file processor as specified is fairly insensitive to other perturbations in the network.

VI. CONCLUSIONS

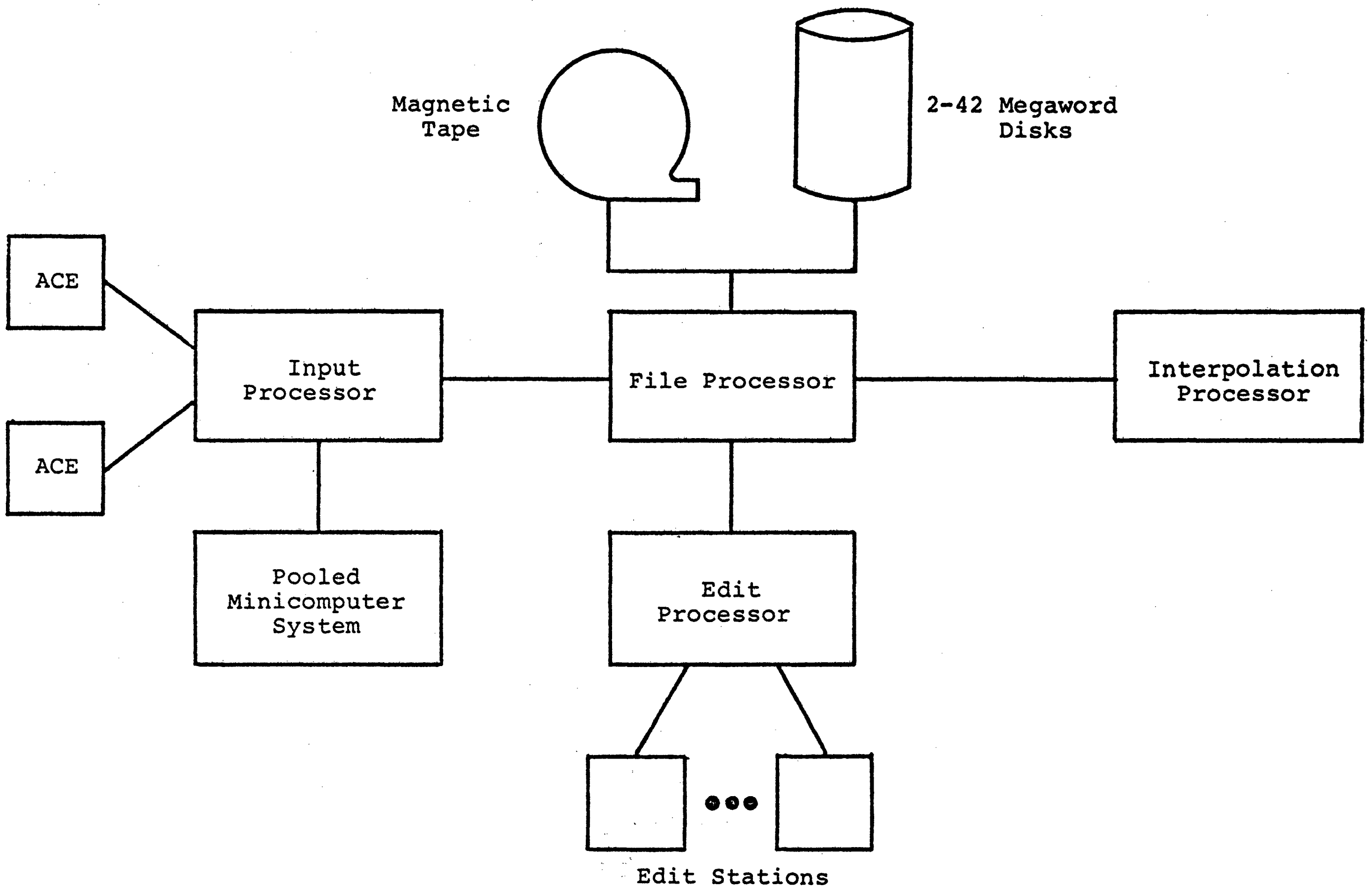
We have described the design of a distributed computer network dedicated to solving a specific real time production problem. The network may be viewed as a "star" architecture of homogeneous processors including a central processor for message routing. The design is based on the concept of functional partition of the necessary computational tasks and fixed assignments of these tasks to individual processors. We have tried to emphasize the total systems approach taken in solving this problem.

REFERENCES

1. G. A. Anderson and E. D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples," ACM Computing Surveys (December, 1975), pp. 197-213.

2. G. M. Elphingstone, "Photogrammetric Collection Techniques for Digital Terrain Data," 1976 ASP Spring Convention, Washington, D.C.

3. R. D. Olsen, "Analytical Stereoplotters in a Distributive Computer Network (Pooled Minicomputer Systems)," 1975 ASP Fall Convention, Phoenix, Arizona.



DISTRIBUTED COMPUTER NETWORK

Fig. 1.

APPLICATION OF A PARALLEL PROCESSING COMPUTER IN LACIE

Sherwin Ruben
Goodyear Aerospace Corporation
Akron, Ohio

Rudolf Faiss
Goodyear Aerospace Corporation
Akron, Ohio

John Lyon
NASA Johnson Space Center
Houston, Texas

Matthew Quinn
NASA Johnson Space Center
Houston, Texas

Abstract - The application of a programmable parallel processing computer to the reduction of remotely sensed multispectral data from a satellite is discussed. Significant performance advantages are shown when compared to a previously employed serial computer in a production environment. Additionally, parallelism of the device allows ready exploration of novel approaches to image processing. The programmability permits diverse exploitation of a large data base and rapid computational capabilities in research applications.

Introduction

The Large Area Crop Inventory Experiment (LACIE) is a joint investigation by NASA, USDA, and NOAA to determine the usefulness of computer-analyzed remotely sensed data in crop forecasting on a global scale. A sampling of LANDSAT imagery selected as representative wheat-growing regions, NOAA-supplied meteorological data, and ground truth history are combined to make predictions on crop yield. The Johnson Space Center (JSC) role in LACIE includes implementing a Classification and Mensuration Subsystem (CAMS), which performs traditional pattern recognition processing on the LANDSAT imagery. The CAMS is an extension of a previously developed JSC software/hardware system, the Earth Resources Interactive Processing System (ERIPS), Ref. 1 and 2, of somewhat more general applicability. The CAMS is tailored to the production problem presented by LACIE requirements to classify large numbers of fundamentally similar regions in the same manner. In short, LACIE (and CAMS) represent an essential change from R&D to a near-production environment.

The purpose of this paper is first to describe ERIPS briefly; second, to discuss why the system was changed to include a parallel processor; third, to describe the processor selected; and fourth, to show a few results.

Since 1972, the ERIPS has been resident on one of five IBM 360/75 computers in the real-time computer complex in the Mission Control Center. Contention for this processing resource with manned space flight support functions has historically been a constraint to ERIPS users. Predictions at LACIE conception indicated that some 40 to 60 hours a day of central processor availability would be needed for maximum project loads, which was clearly incompatible with ground support shuttle program development to be performed within the complex. Alternative processing techniques and/or equipment were sought under the additional constraint of schedule and consequent desirability of retaining the ERIPS software and equipment structure to the maximum practical extent. Uncertainties in the final nature of the LACIE problem presented the possibility of additional computationally bound routines, further constraining legitimate solutions by precluding consideration of hardwired equipment already in use in some facilities for treatment of ERIPS-like algorithms. It became necessary to consider acquisition of a fully programmable computer system having parallel or pipelined processing capabilities that would provide an increase in throughput while reducing the burdens on the 360/75's.

Management of the LACIE data base, consisting of some 4.2 billion bytes of disk storage, and the complex software needed to interface between the user and application software, as well as development continuity and schedule, demanded retention of the 360/75 as the ERIPS/LACIE supervisor. Furthermore, the existing ERIPS system is used as the heart of an expanding LACIE data handler. The system configuration dictated under the above requirements is shown in Figure 1. The parallel processor selected was the Goodyear Aerospace STARAN,[a] hereinafter referred to generally as the "SPP" (special purpose processor). The remainder of the paper briefly describes the SPP, the pattern recognition algorithms implemented thereon, and results and conclusions to date regarding the operational system.

STARAN Computer Description

The SPP STARAN system (Ref. 3 through 6) is based on a computer organization in which many identical operations are executed simultaneously; that is, it is a "single instruction stream, multiple data stream" processor. For example, in the SPP an "add" operation can be executed simultaneously for 512 pairs of numbers. The parallel execution of an operation for many data pairs is made possible by employing many processing elements (512).

[a]Trademark, Goodyear Aerospace Corporation, Akron, Ohio 44315.

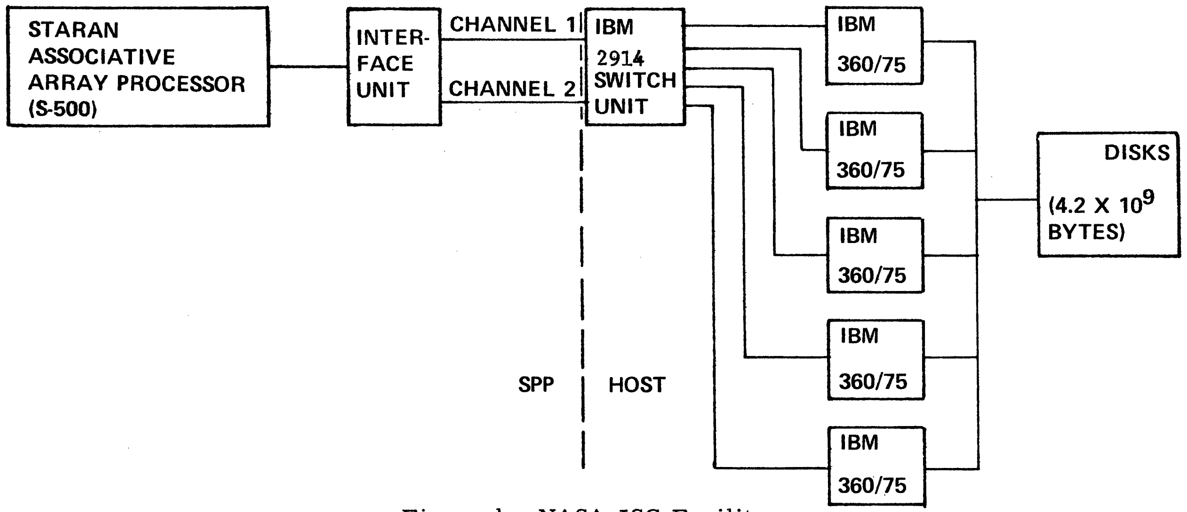


Figure 1. NASA JSC Facility

A top-cut diagram of the SPP main frame is shown in Figure 2. It consists of a conventionally addressed control memory for program storage and data buffering, a control logic unit for sequencing and decoding instructions from control memory, and two associative array modules.

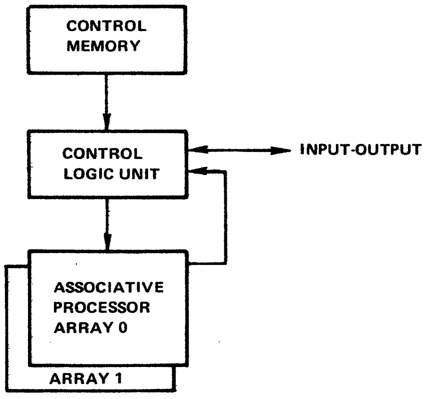


Figure 2. SPP Top-Cut Architecture Diagram

The high processing and throughput speeds that the SPP achieved resulted from the unique capabilities of the associative array (Figure 3). Each SPP array contains 256 simple processing elements. All processing elements (PE's) perform the same operation at the same time, but each processing element acts on independent data. Thus, in each SPP array, 256 independent data streams can be processed simultaneously. For the two array SPP system, 512 independent data streams can be processed. Only two of a possible 32 arrays were needed to achieve the required processing rates demanded for LACIE; processing power growth capability of 16 to 1 is possible.

Array memory used to support the PE's is comprised of 256 words having 256 bits. Multiple access paths between the PE's and the bit memory locations provide ready access to 256 different bit patterns in the array. Two access "stencils" are shown in Figure 3.

To further enhance the data routing capability of an array module, an alignment, or permutation, network in the machine provides a flexible interconnection between processing elements.

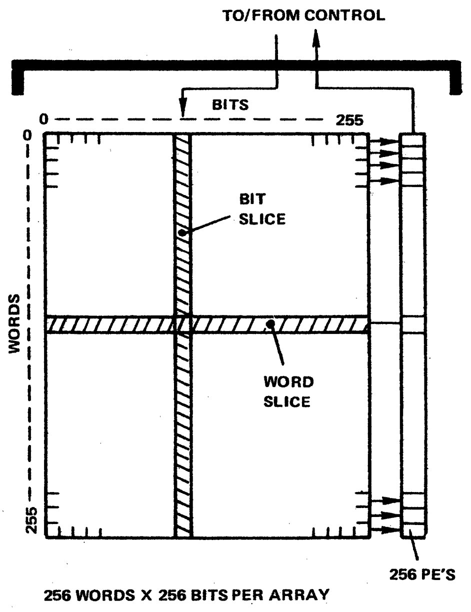


Figure 3. SPP Associative Array

The multiple processing elements, the multidimensional access memory, and the permutation network give the SPP the flexibility to be useful for a wide set of problems.

LACIE Algorithm Execution

Algorithm Description

NASA is using the SPP in the LACIE program for pattern recognition functions. The SPP performs such processing tasks as statistics, iterative clustering, adaptive clustering, maximum likelihood classification, and mixture density classification.

The algorithms are all well suited to the SPP architecture because they have an inherent parallelism resulting from a given computation being performed on all picture elements (pixels) in an image. Since computation associated with each pixel is the same for a given algorithm, it can be implemented in a single-instruction stream. The LACIE algorithms thus fit the single instruction, multiple-data stream concept that is part of the SPP architecture.

Statistics Calculations. The statistics calculation algorithm develops statistical data that characterizes a group of measurement vectors that have been assembled. The statistical data developed for the measurement vectors of the group include vector component mean and covariance values.

Iterative Clustering. The iterative clustering algorithm provides a means both for assigning measurement vectors to clusters and for evolving the statistical description of the reference clusters. The algorithm determines the "distance" of each measurement vector (of a set of such vectors) from the mean vector of each cluster and assigns each measurement vector to the "nearest" cluster. The statistics of all measurement vectors assigned to a particular class are determined and are used to modify the original clusters and cluster statistics. When the tasks described above are accomplished, the algorithm is considered to have undergone one "pass." Usually, several passes are executed before the iterative clustering process is terminated.

Adaptive Clustering. Like the iterative clustering algorithm, the adaptive clustering algorithm provides a means of grouping similar measurement vectors (similarity is determined by closeness in an N-space). Unlike the former algorithm, no a priori knowledge is required to "prime" the algorithm.

Maximum Likelihood Classification. The objective of the classification tasks is the final assignment of a measurement vector to a defined class. The processing functions described previously are designed to obtain, refine, and normalize input class statistical measures and to create new classes as necessary for reference input to the classifiers.

The maximum likelihood classification algorithm involves essentially the calculation of the function representing the probability that a given vector belongs to a class and the determination of the most likely class among those defined for the vector.

Mixture Density Classification. The mixture density classification algorithm is similar to the maximum likelihood algorithm. The distinction is a derivative of the class statistics definition made in each case. The maximum likelihood classification algorithm utilizes a set of class statistics (mean and covariances) obtained for the population of the class as a whole; the mixture density function is formulated to treat a class as a union of independent subclasses, each of which is described as a population having a complete set of (sub-) class statistics. This representation tends, under careful preprocessing and definition of subclasses, to separate a population consisting of a multimodal distribution into several unimodal distributions and to improve the performance of the classification algorithm.

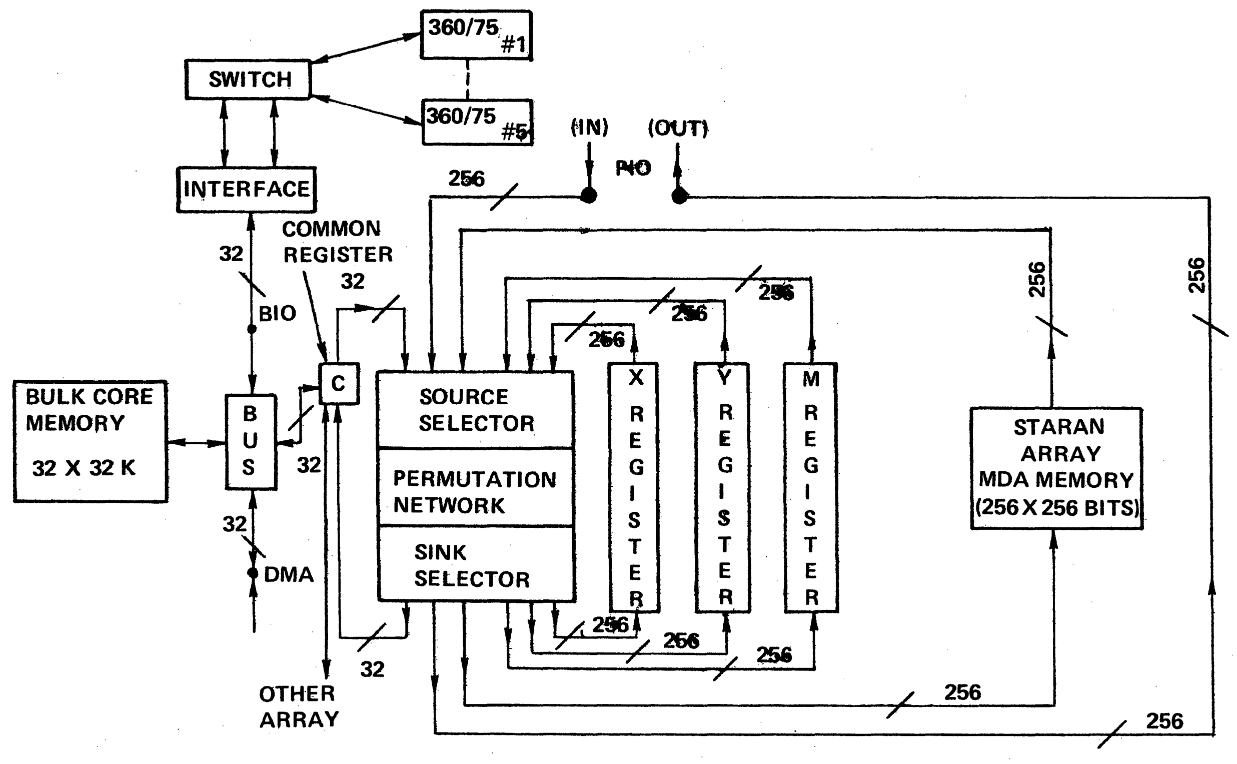
SPP Resource Utilization

SPP-to-Host Connection. As shown in Figure 4, a number of different paths exist for moving data into and out of the SPP. From a user's standpoint, the primary difference between the paths is the rate at which data may be moved. The entry path into the SPP via the SPP's sequential controller provides an I/O rate on the order of one megabyte per second. The fastest entry path into STARAN is the PIO (parallel input/output) path, which can support data rates on the order of 80 megabytes/second/array. Midrange rate paths into STARAN that are available are a DMA (direct memory access) path and a BIO (buffered input/output) channel. Both paths support transfer rates on the order of two megabytes/second.

For the LACIE program, the BIO path to the SPP was chosen for moving data to and from the SPP; that is, the BIO channel was connected to one of five 360/75's via a custom-built interface unit. The BIO entry path was chosen because it could meet LACIE data transfer rate requirements of about 100K to 200K bytes/second from the host. Also, peak rates as great as 0.5 to 1.0 megabyte/second could be supported by the channel.

In practice, when one of the five pattern recognition processing tasks is requested to be performed, input vector data is moved to the host output buffer region. The interface passes this data to a corresponding receiving buffer on the SPP side that is defined by the SPP application program. The movement of data between the SPP and the connected host is invisible to the SPP application programmer, and application tasks are able to be executed concurrently with I/O operations.

Application Program Executive. Although the STARAN is a stand-alone computer system, the SPP acts as a slave to the host in the LACIE. Application program processing can only be initiated from the host side of the interface unit. The SPP, upon completing a task, waits for the next task request from a connected host. The request occurs in the form of a data block sent by the host to the SPP. The SPP applications executive program passes out of a wait loop when the transfer is complete, interprets this block, checks for errors, and then initiates the transfer of the requested task application program from SPP disk to control memory.



X, Y, AND M REGISTERS ARE THE PROCESSING ELEMENTS

Figure 4. SPP Primary Data Paths

Input Vector Data Transfer. The SPP's control memory (CM) is set up to receive blocks of 1024 input vectors. The data is stored on a component ("channel") basis; thus, the memory required for a particular component of each of the 1024 vectors is 256 32-bit words since each component is specified to be represented by an unsigned 8-bit positive number. To support the processing strategy that evolved for LACIE, it is necessary to move this data into the SPP arrays in the configuration shown in Figure 5. In this configuration (used for all five application tasks), one input measurement vector is assigned to one SPP array word location.

The common register is repeatedly loaded from CM and stored into the 256-bit-long "X" processing element (PE) register until the X register is full. When full, the 256-bit X register is dumped into its own array in 265 nanoseconds using the 8-word × 8-bit array access mode (one of 256 access modes). The cycle is repeated until a particular component field is loaded and until all component fields are loaded.

Data Ordering. When using the 8 word × 8 bit array access mode to store to the array from CM, the order of the 512 vectors loaded is scrambled. For all LACIE tasks that do not require the computation of statistics for a vector set, such scrambling has no impact because processing steps are performed on a per pixel basis. Thus, all processing steps for a given vector involve only field locations in the one word assigned to it. Ultimately, the output vector data produced as a result of processing (e.g., the classification index for the vector when a classification task is called) is located in the word associated with the vector. As a result, data transfer to control memory uses the inverse steps used in the data transfer from control memory. Writes become reads, reads become writes, etc. The same 8-word × 8-bit array access stencil is used for both directions of data movement.

The statistics task requires an ordered data base. When this task is executed, the goal is to achieve the statistical characteristics of the set of vectors that are found within known geographical boundaries (test fields). This data is always sent to the SPP as a contiguous set. No set labels are shipped with the data, and so order and a vector count for each set are used to distinguish vectors of one set from those of another set. After the statistical quantities associated with the individual vectors are produced, they must be summed over the set. The across-vector summing procedure requires that data for the vectors of a set lie in contiguous words within the arrays.

Ordering of 512 8-bit items requires less than 8 microseconds. Such rapid ordering execution times are possible because of the flexible routing capability of the routing network associated with each array and the high bandwidth path to the array memories.

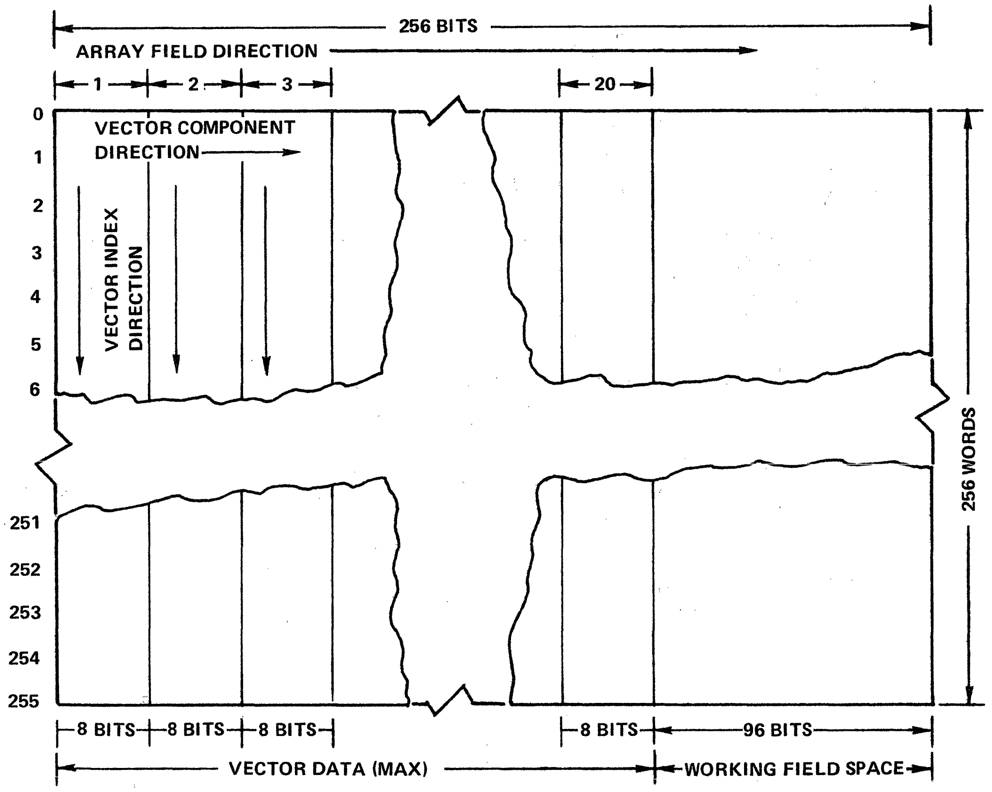


Figure 5. Measurement Vector Layout in STARAN Array

LACIE Word-Oriented Arithmetic Operations (Conventional). During the execution of a LACIE task, measurement vector data loaded into the SPP array fields may be subjected to one or more of the following assembly language supported array arithmetic operations: (1) field-to-field add, subtract, or multiply; (2) field-to-common add, subtract, or multiply; and (3) field absolute value.

Initially, the 8-bit-wide vector component data is unsigned; the STARAN assembly language (APPLE) does not support unsigned operations. Thus, all component data were biased down by 128. Then the value of each component lies in the interval from -128 to +127, inclusive, a number range that is accommodated using an 8-bit field that includes a sign bit slice. It was particularly simple to offset the data since it only required that the most significant bit of each field be complemented. At the expense of 125 nanoseconds, each lead bit slice of a component field is complemented as it is loaded into the X register of the processing register group and restored into the same bit slice of the arrays (at the expense of 265 nanoseconds). Thus, biasing operations expend about 8 nanoseconds/vector component.

For the statistics task (or for statistics-type processing performed inside various variations of the iterative clustering task), covariance values need to be computed. The first step performed to determine the covariance values for a set of vectors is to find all cross products of the components of each vector. These multiplies are performed during task execution using standard STARAN array field*field multiply routines; answers are placed into a 16-bit scratch field to await additional across-word processing. To get a component field times a component field cross product, an 8 bit $\times$ 8 bit field multiply is executed. Such an operation expends about 50 microseconds (or about 100 nanoseconds/component pair).

It should be noted that the SPP uses no multiply hardware. Multiply times are dependent on a software program. It can be shown that field*field computation times are directly proportional to the product of the number of bits in the multiplier and multiplicand. It is clear that, to achieve highest multiply execution rates within the SPP, field lengths must be minimized. Since the APPLE assembly language was designed to accommodate arbitrary field lengths and arbitrary field starting locations, it provides convenient means to exploit any reduction in field length that can be justified by physical problem constraints.

A particularly straightforward example for such exploitation occurs in the "assign measurement-vector-to-cluster" phase of the clustering tasks. The "distance" of each measurement vector, $\left(x_1^p, x_2^p, \text{-----}, x_n^p\right)$, from a cluster center (as defined by a cluster mean vector, $\left(\mu_1^c, \mu_2^c, \text{---}, \mu_n^c\right)$, is determined according to the distance, $d_{c,p}$, definition:

$$d_{c,p} = \sum_{i=1}^{N} \left| x_i^p - \mu_i^c \right| ,$$

where p is the pixel index, c is the cluster index, i is the vector component index, and N is the dimension of the vector space. Each component value x_i^p is defined as an 8-bit signed integer.

The following describes how the distance computation is achieved. The μ_i^c statistics are received from the host as single precision floating point numbers. Within the SPP, the μ_i^c values are converted to fixed point (23 bits), biased, sign changed, and then stored as $\left(-\mu_i^c\right)$ within the SPP high-speed data buffer segment of control memory. The operations described are accomplished as part of the initialization operations for the clustering task. Distance measurements for all of the 512 20-component measurement vectors are calculated in one millisecond.

Since 30,000 arithmetic operations (adds, absolute values) are performed when finding vector-to-cluster distance, the time to execute one "average" operation is about 35 nanoseconds. If μ_i^c were reduced to 13 bits, this execution time would be nearly halved.

Word-Oriented Arithmetic Operations (Special). A constant concern that existed in the SPP software design phase of the LACIE program was related to array-memory resource management. Since the SPP has 256 bits/word and since provision must be made to hold a maximum of 20 vector components inside the array, only 96 bits of field space remained to accommodate various scratch storage and processing storage fields. The price of ignoring the array storage constraints would have been costly from the standpoint of program execution time since data that could not be stored within the arrays would have been required to be swapped back and forth between control memory and array memory via the common register funnel.

A two-pronged strategy was pursued in the effort to hold down field space requirements for classification tasks.

First, the classification algorithm was examined to see whether constraints could be imposed on the size of the numbers encountered in generating the maximum likelihood pixel-to-class assignment confidence number. The original confidence number for a pixel, p, tested against a class, c, namely, $h_{p,c}$ was described by

$$h_{p,c} = K^c + 1/2\,(X^p - U^c)'\,\Gamma^{c-1}\,(X^p - U^c) ,$$

where Γ^{c-1} is the inverse covariance matrix for class c, U^c is the class c mean vector, X^p is the measurement vector associated with pixel p, and K^c is an a priori established biasing constant.

In investigating the constraints on the range of values of the elements of Γ^{c-1}, it was observed that the inverse covariance matrix could be expressed as a product of a lower triangular matrix and its transpose. $\Gamma^{c-1} = L^{c'}\,L^c$, and so the expression for $h_{p,c}$ could be expressed in terms of the inner product of the vector $R^{p,c} = L^c(X^p - U^c)$; that is, as

$$h_{p,c} = K^c + 1/2\left[L^c\,(X^p - U^c)\right]'\left[L^c\,(X^p - U^c)\right] .$$

The theoretical importance of this formulation is the fact that it can be proven that the components of $R^{p,c}$ have a variance of 1 when a pixel is indeed a member of the class c. The very necessary condition for managing array field sizes within the SPP - a constraint on the maximum number size - was present.

The form for $R^{p,c}$ was massaged further to put it in the form

$$R^{p,c} = L^cX^p - L^cU^c .$$

The leftmost product in $R^{p,c}$ above is both pixel and class dependent and so will change for each pixel; the rightmost product is strictly class data dependent, and the components of this vector need be generated only once for each vector, independent of the number of pixels that need to be classified.

The importance of this form, when using the SPP's architecture, cannot be overemphasized for it suggests a different processing order from that suggested by the earlier form. The earlier form suggests performing the (X^pU^c) subtraction first and the matrix/vector multiplication second. In the LACIE classification programs, the class mean vector components were defined as 8-bit signed integers with 15 fractional bits; the X^p components are 8-bit signed integers. The difference vector would require 24 bits. Thus, if the L matrix elements are described by 24-bit signed fractional bits (after normalizing so that the largest element of L is set to lie in the interval of 0.5 to less than 1), then the L times difference vector multiply operation requires N(N + 1)/2 24-bit*24-bit common-times-field multiply operations (where N is the dimension of the vector).

The latter form suggests performing the matrix measurement vector multiply operation first and then subtracting the L^cU^c vector. Such an order of processing requires N(N + 1)/2 multiplies, as before, but the multiply operations are 8-bit* 24-bit common-times-field multiplies. The latter multiplies will be executed nearly three times faster than the former multiplies. Furthermore, the latter product conserves field space much better than the former procedure, since a much shorter product field (32-bits) is produced.

Despite the use of constraints to minimize field widths, the 96-bit-wide available field space proved to be too small to contain all the fields required by APPLE to perform the algorithm. Thus, the strategy of writing special subroutines to reduce field space was employed. In particular, a special routine was written to produce the ith element of $L^c X^p$; namely, the sum

$$\sum_{j=1}^{N} \ell^{c}_{i,j} x^{p}_{j} ,$$

where $\ell_{i,j}$ is the ith row jth column element of L^c and x_j is the jth row element of the vector X^p. The routine adds the product directly to the accumulation field and so by-passed the need for a product field space allocation.

A second special routine was written to conserve field space when squaring the $R^{c,p}$ element. Rounding was introduced within the squaring operation so that the square field and $R^{p,c}$ element field could both overlay the accumulate field previously discussed. A side benefit of the squaring routine is that it executes about twice as fast as an equivalent field*field multiply operation even when no rounding operations are required.

The dual strategy for managing field space in the maximum likelihood classification program was successful; no control memory was required to hold intermediate results when executing this task. For the mixture density task, it was not possible to preserve enough field space for large dimensioned vectors and so the control memory had to be substituted for array memory when vector dimensions became sufficiently large. Nevertheless, the array field management procedure reduced the need to access control memory.

Across-Word Arithmetic Operations. Arithmetic operations discussed (whether conventional or special) were all performed within words. Only the X, Y, and M PE register bits associated with a word were involved in executing such operations. Yet, when performing statistics processing tasks, it is clearly evident that statistical entities must be added across words. Thus, to get the sum of first components of a set of vectors within the array, all items within an interval of a field column must be added together. To support such requirements, a special vertical add routine was developed. To use the routine, an auxiliary bit slice that marks the end of a vertical group of entities must be available. The routine adds all items between end marks and places the sum in a field (specified by the calling sequence to the routine) adjacent to the end marks of a logical group. The routine makes extensive use of the shift capability of the STARAN routing network; because of it, the routine can typically add vertical groups totalling 256 16-bit items (one group per array), grouped arbitrarily, in about 100 microseconds.

Results

General Commentary

The LACIE performance advantages of the SPP over the previous 360/75 are functionally dependent upon: (1) algorithm organization (the ability to exploit parallelism); (2) number of data channels; (3) number of signatures (classes/clusters); (4) number of pixels (vectors) per quantum of system workload (job); (5) SPP setup time (formatting of vector transfers to and from the SPP); and (6) data base retrieval rates.

The effects of these drivers are mutually dependent and difficult in many cases to distinguish. The sampling of results provided below will be generally treated in terms of these driving functions, with only a few specific comments in order, as they relate to computational idiosyncracies of the individual algorithms. Some preliminary remarks:

First, in general, a LACIE image consists of 22,932 data vectors or up to four such sets of vectors. The number of channels (dimensionability) ranges between 1 and 20, although in practice the pattern recognition processes in the production system are executed normally on 4, 8, 12, or 16 channels. A maximum of 60 signatures for classification may be defined; practically, this value remains ordinarily between 10 and 30. Other system delimiters, as described under "LACIE Algorithm Execution," are generally exploited operationally across their entire range. Extensive testing of the SPP software in the production environment confirmed both logical and performance timing behavior of the system throughout the range of software specifications.

Second, the historical driver of the 360/75-based LACIE/ERIPS performance was the CPU. In the SPP configuration, principal limitations on throughput are, in practice, the retrieval functions from the imagery storage medium, the IBM 2314 disks. Only on jobs of significant complexity, specifically classification exercises on 12 channels or greater with discrimination of more than 20 classes, does the system perform in an SPP CPU-bound state. Development of an imagery data retrieval technique (Ref. 7) has ensured optimal exploitation of the disks for the peculiarities of the LACIE application, but the disks generally remain the system driver. Direct access to the imagery on the ITEL 7330 data base would permit significant throughput improvements for most LACIE jobs; such implementation may be made at a later date, as necessary, but current performance (although suboptimal because of I/O) satisfies existing resource constraints.

Third, as discussed previously, SPP arithmetic is field-length dependent in performance characteristics. The LACIE applications specifications dictated effective equivalence with 360/75 floating-point arithmetic results for purposes of continuity; this stringent requirement on the SPP, which was achieved, is not statistically justified on the basis of measurement vector variance, and legitimate results of processing can be obtained via shorter fields than employed with significant performance advantage.

Fourth, in a comparison of pre- and post-SPP timing, the control base was modified to some extent in software that could have affected 360/75 applications performance; that is, certain 360/75 system software routines were optimized at the time of SPP implementation. These changes could, to some extent, be reflected in the timing figures given below for pre-SPP algorithms, but the figures shown display pre-SPP results without such system changes. Further, the adaptive clustering algorithm was extensively and theoretically modified when incorporated into the SPP; the objective was to maximize the benefits of parallelism and to utilize spatial as well as spectral data characteristics. The result has been a technique of improved convergence and stability, but no direct (timing) performance comparisons can be made with pre-SPP results.

Statistics

Statistical processing ordinarily occurs fairly rapidly in the LACIE system and was included in the SPP development for consistency with the notion that all pattern recognition processors of a pixel-dependent type would be SPP-resident. Also, the STATS routine is invoked in the body of ITCLUS; SPP implementation reduced organizational complexities. LACIE characteristics, however, include occasional and numerous small (<20 pixel) fields on which processing must be performed; SPP performance is severely compromised via system overhead on such jobs. Occasionally, SPP STATS is slightly slower even than the 360/75 STATS, but has never been less than 90 percent of 360 rates (on tasks of four to five seconds). On larger fields and on large channel set jobs, the SPP performance advantage reaches about 3 to 1, but 360/75 execution would not be deleterious to the system because the process rarely requires more than 20 seconds on the 360 in the most complex LACIE cases.

Clustering

An adaptive/iterative clustering exercise was defined for a benchmark as follows: 500 X 200 (10^5) vectors, 16 channels, to be distinguished into 10 clusters in an artificial data set. Results: non-SPP required 35.1 minutes, SPP required 37 seconds, a performance gain of 57 to 1.

Figure 6 shows typical LACIE results for 22,932 vectors, under various channel set sizes and (implicity) discriminated clusters. Performance gains are less than for the benchmark, reflecting system overhead penalties for smaller data sets, but demonstrating the I/O constraints driving the SPP on complex applications and significant performance improvements (up to 15 to 1) normally experienced.

Classification

A classification benchmark was defined as follows: MAXLIK, 4 channels, 10 classes, 2340 X 3240 vectors (7.58 million pixels). Results: pre-SPP, 105 minutes; SPP, 8.15 minutes, a performance gain of 13 to 1.

Figure 7 shows MIXDEN results on LACIE images of 22,932 vectors under various channel set sizes and 20 defined signatures. As in clustering, system overhead diminishes performance factors on smaller segments of data, although the trends are clearly I/O driven. MAXLIK, organizationally essentially identical to MIXDEN, produces timings approximately 20 percent less for both SPP and non-SPP.

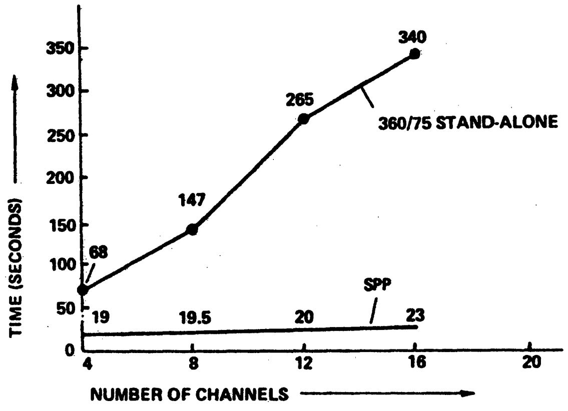


Figure 6. Iterative Clustering Timings

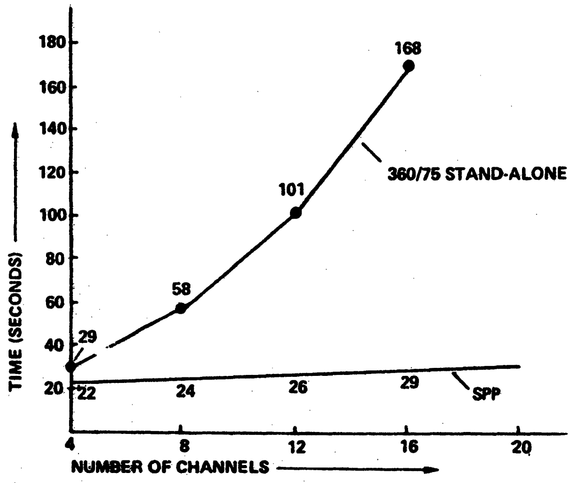


Figure 7. MIXDEN Timings

Conclusions

The SPP has satisfied and exceeded performance specifications originally defined. The system performance can be significantly improved, when necessary, by modifications in the host data retrieval technology without impact to the SPP software or addition of arrays. Within the LACIE context, the most tangible improvements have been in processes (clustering, classification) that were previously prohibitively expensive users of host resources. Due to host I/O constraints, the statistics function on the SPP, as anticipated, offered relatively little improvement except in exotic test cases involving large data sets.

Additionally, the SPP affords users of earth resources remote sensing technology access to computationally feasible spatial/spectral data analysis techniques (e.g., adaptive clustering) that have heretofore been clumsy or burdensome on serial devices. Extensions and modifications to this methodology are in progress, for investigative and possible production purposes.

As anticipated prior to the SPP procurement, additional requirements, both modifying existing algorithms and proposing entirely new analytic techniques, are currently in development in LACIE as SPP functions. These schemata, including "iterative" classifiers and several varieties of temporal change classifiers, previously have been possible only on limited amounts of data due to serial device limitations. Access to the large LACIE data base and the performance improvements of the SPP are permitting extensive study of these techniques prior to production system inclusion.

In summary, the LACIE environment, including high throughput requirements in a quasi-production system and a requirements flux in a technologically and theoretically developing discipline, has demonstrated the cost-effectiveness and utility of a programmable SPP. We believe that this utility will continue for several years, and particularly that this essentially research-oriented system will offer highly beneficial guidelines toward the development of true production systems, for agricultural and other purposes, employing multispectral scanning data.

References

1. NASA/JSC, ERIPS Requirements, Change 6, Document JSC-10152 (SISO-TR-514), Nov. 1975.

2. IBM Federal Systems Division, Houston, Texas, Large Area Crop Inventory Experiment (LACIE) User's Guide, Revision 6, 27 Feb. 1976.

3. K. E. Batcher, "The Flip Network in STARAN," 1976 International Conference on Parallel Processing, Aug. 1976.

4. K. E. Batcher, "The Multi-Dimensional Access Memory in STARAN," 1975 Sagamore Computer Conference on Parallel Processing, p. 167.

5. L. A. Gambino and R. L. Boulis, "STARAN Complex - Defense Mapping Agency, U.S. Army Engineer Topographic Laboratories," 1975 Sagamore Computer Conference on Parallel Processing, pp. 132-141.

6. E. W. Davis, "STARAN Parallel Processor System Software," 1974 National Computer Conference, AFIPS Proceedings, Vol. 43, pp. 17-22.

7. A. E. Pape and D. L. Truitt, "The Earth Resources Interactive Processing System (ERIPS) Image Data Access Method (IDAM)," Symposium on Machine Processing of Remotely Sensed Data, 29 June, 1976. Purdue University, West Lafayette, Ind.

HIGH-RESOLUTION IMAGE PROCESSING ON A PARALLEL COMPUTER SYSTEM (a)

W. W. Gaertner, M. P. Patel, S. S. Reddi, C. T. Retter and I. M. Singh
W. W. Gaertner Research, Inc.
205 Saddle Hill Road
Stamford, Connecticut 06903

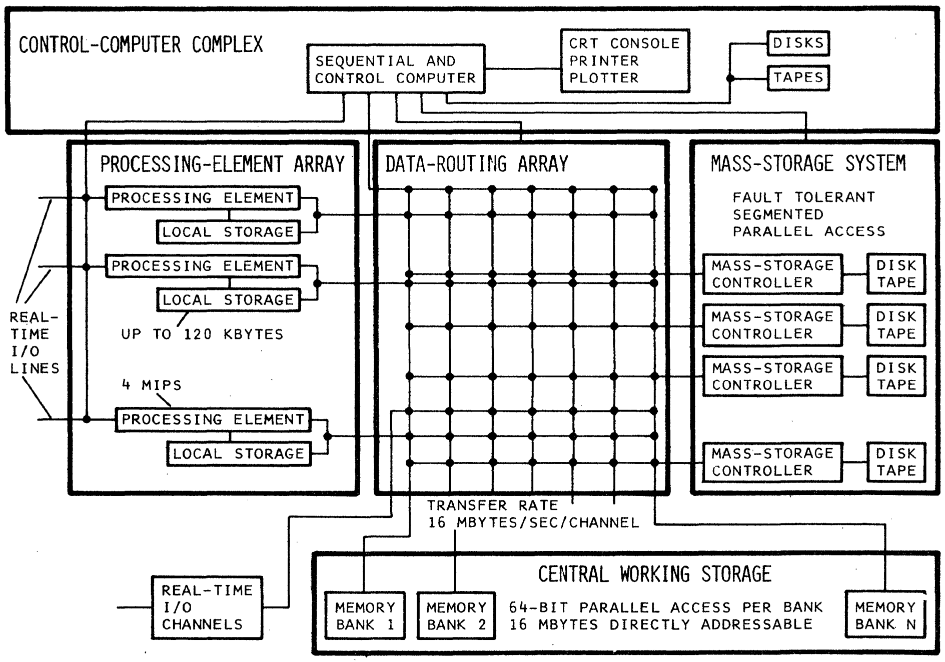
SUMMARY

A simple frequency-domain filter operation on a 1024 x 1024 pixel image requires approximately 80 million real floating-point multiplications. Image-processing at rates of approximately 1 frame/second is therefore beyond the reach of any sequential computer. Taking advantage of the high degree of parallelism inherent in all image-processing algorithms, a parallel computer architecture, the G-471, has been developed (see W. W. Gaertner, "Architecture for a Highly Reliable Parallel Computer System", Proc. 1975 Sagamore Computer Conference on Parallel Processing, p. 125) which consists of an array of floating-point hardware-enhanced microprocessors and a large multiported common memory under the control of a sequential computer. A typical configuration as shown in Figure 1 achieves 100 MIPS and contains 16 Mbytes of 500 ns memory. This paper analyses the parallelism in such image-processing algorithms as two-dimensional Fourier transforms, table look-up filters, low-pass, high-pass and band-pass filters, homomorphic filters, constrained least-squares filters, Wiener minimum mean-square error filters, parametric Wiener filters etc., and presents the equations which determine the number of additions, multiplications, divisions, log and exp operations to be performed, as well as the amount of high-speed memory required to hold interim results during processing.

It is shown that a large memory bandwidth between the processing-element array and the mass memory is as important to the throughput as the processing power of the processing elements themselves.

Finally, the relationship between throughput and hardware costs is derived, leading to the conclusion that, in image processing, a computer of proper architecture can have a performance/cost ratio 2 orders of magnitude higher than that of a large sequential computer.

Figure 1



Block Diagram of the G-471 Parallel/Associative Computer System

W.W. GAERTNER RESEARCH INC.

(a) This work was sponsored by DARPA and RADC.

AN IMPLEMENTATION OF THE HADAMARD TRANSFORM ON THE STARAN ASSOCIATIVE ARRAY PROCESSOR

Annette J. Krygiel
Defense Mapping Agency
Aerospace Center
St. Louis, MO 63118

Summary

The Hadamard Transform, if performed in a straightforward manner, requires N^2 additions/subtractions for the one-dimensional case, where N equals the number of data points. A number of authors [1, 2, 3, 4] have provided computational algorithms for a fast Hadamard Transform (FHT), requiring $N \log_2 N$ additions/subtractions. These algorithms have been implemented on a variety of sequential processors. The implementations vary in certain characteristics.

However,

(1) Their basic approach is analogous to the method of the Cooley-Tukey fast Fourier transform (FFT), typified by the FFT butterfly, with replacement of the multiplication factors by the ± 1's of the Hadamard matrices.

(2) Even though the algorithms differ in speed, they are all θ $(N \log_2 N)$.

A decimation in frequency Hadamard butterfly can be described as:

$$X_{m+1}(p) = X_m(p) + X_m(q)$$

$$X_{m+1}(q) = X_m(p) - X_m(q)$$

where X = input signal vector of N points
m = iteration level
p, q index the pairing of data so that the separation of points is $N/2^m$.
There are N/2 butterflies for each level and $\log_2 N$ levels giving θ $(N \log_2 N)$ operations.

Using a similar algorithmic approach but employing a parallel processor operating on N data points simultaneously, a reduction in computation time on the order of N should be achieved, i.e., $\theta(N \log_2 N) \rightarrow \theta(\log_2 N)$.

An FHT was implemented on a standalone four array STARAN at the DMA/ETL Facility [5]. The algorithm is a one-dimensional decimation-in-frequency FHT subroutine operating on a maximum of 1024 16 bit data points; the original vector and intermediate results are destroyed. In all cases, N processing elements are used. The arithmetic is fixed integer.

Data is moved into the arrays; then, for each iteration, every point has its copoint on the butterfly positioned alongside. This is done using a columnwise rotation when $N/2^m \leq 256$; otherwise through the appropriate array to array movement of data. The required data fields are complemented, and then addition (or subtraction) of the pair transpires. While the objective would be to complete each iteration in one arithmetic operation, several sources of overhead exist:

- proper pairing of the data points including setup for inter-array movement for the 512 and 1024 point cases.
- complementing half the data fields to avoid a two operation penalty - N/2 additions followed by N/2 subtractions - so that a single parallel addition on N data points is sufficient.
- testing N for appropriate actions commensurate with the foregoing.

Execution times attained for the FHT on STARAN using Page Memory and the High Speed Data Buffer are:

N	Arrays	Time*
64	1	288μs
256	1	380μs
512	2	459μs
1024	4	538μs

*Times do not reflect set-up for moving data to the arrays or mask generation for complementing.

Placing the FHT subroutine in Bulk Core degrades speed by $\simeq 2.5X$. Normalizing by N after two calls of the subroutine requires 10-12μs.

Results are impressive; an order N reduction is not achieved primarily due to the slow bit serial nature of STARAN arithmetic. The 1024 STARAN FHT is projected at a 70X improvement over a FORTRAN FHT [3] implemented by the author on UNIVAC 1108.

References

[1] William K. Pratt, Julius Kane, Harry C. Andrews, "Hadamard Transform Image Coding", Proceedings IEEE, Vol. 57, (Jan 69),pp 58-68.

[2] L. J. Ulman, "Computation of the Hadamard Transform and the R-Transform in Ordered Form", IEEE Transactions on Computers, (Apr 70), pp 359-360.

[3] Vijay K. Agarwal, "A New Approach to the Fast Hadamard Transform Algorithm", IEEE Computer Group Repository, R-70-2043, (1970).

[4] M. Kunt, "On Computation of the Hadamard Transform and the R Transform in Ordered Form", IEEE Transactions on Computers, (Nov 75), pp 1120-1121.

[5] L. A. Gambino, R.L. Boulis, "Defense Mapping Agency/USAETL STARAN Complex", Sagamore Computer Conference Proceedings, (Aug 75), pp 132-141.

ON THE FORMAL DEFINITION OF PROCESSES

Pamela Zave
Computer Science Department
University of Maryland
College Park, Maryland
20742

Abstract -- The only model of logically concurrent, asynchronously interacting processes which has had real impact on the design and implementation of processes is that of programs in execution which communicate through shared variables. It is shown that this model cannot be formalized successfully for general purposes. An alternative model based on message communication, which can be formalized successfully, is proposed. A comparison of the two models with respect to mutual exclusion of concurrent processes sharing a data base is used to argue that the message model is as intuitive as the shared variable model, but richer in the computational structures offered to the designer.

Introduction

The existence and importance of processes as the fundamental dynamic units of computation have been recognized for some time. It is now common for textbooks to discuss logically concurrent, mutually asynchronous processes and their interactions.

It is clear that the only conceptual model of interacting processes which has had real impact on their design and implementation is that of programs in execution which communicate through shared variables. It has led to the definition of powerful programming language primitives such as semaphores ([1]), conditional critical regions ([2]), and monitors ([3]). The purpose of these primitives is to help the programmer use shared variables correctly and conveniently, within the confines of certain common structures.

This paper will argue that a "universal" (in a sense to be defined shortly) formal model of a process is needed, and that concentration on shared variables has been an obstacle to development of a useful one. In the remainder of this section the characteristics and flaws of shared variable communication will be examined, while the third presents an alternative model based on message transmission. In the fourth section the two models are compared with respect to mutual exclusion of concurrent processes sharing a data base.

A universal formal process model is one which can serve as a paradigm in the sense of [4], a conceptual framework in which problems are formulated and solutions are communicated. The partial recursive function is a paradigm for the study of computability and the program is a paradigm for the design of algorithms, but the introduction of concurrency and time-dependence has made both unsuitable as paradigms for dynamic computation.

To be less grandiose, a universal formal process model would be a design tool. It would provide a "language" in which to express ideas precisely at any level of abstraction with unnecessary constraints on lower levels considered harmful. Once a design was formulated, it would be subject to algorithmic analysis, formal proof techniques, optimizing transformations, and other results of theoretical research.

From these uses for the formal model, the criteria of naturalness, usefulness, and generality can be derived. The model should be applicable to familiar situations without undue contortions, it should exhibit properties which facilitate formal manipulations, and it should include the largest possible class of digital phenomena (without sacrificing naturalness and usefulness).

There is certainly no scarcity of formal models of parallel computation ([5], [6], and [7] are good entry points into the extensive literature). These have not been developed in the direction of paradigms, however; rather, the role of a paradigm would be to make the relationships among them, and between them and practical programming, clear.[a]

For example, a Petri net is a good model for studying properties such as correct synchronization and freedom from deadlock. It is possible to represent relevant characteristics of a process design as a Petri net and then verify that it has these properties. But Petri nets will never have much influence on the design of processes because (1) a Petri net represents only a small subset of the properties of a process, and (2) a Petri net is too far removed from the control and data structures of programs, and the allocation of physical processors to them, to guide the designer in constructing a process whose model is a Petri net.

As mentioned before, only the programs-with-shared-variables model has had such influence, and if it could be formalized successfully for general theoretical purposes, that formalization

[a] Actually, many of the relationships among them have been clarified in [7]. It is the relationship between formal models and practical programming that is poorly understood.

would be a strong candidate for paradigm. Unfortunately, any formalization of it[b] is inconsistent with even the loosest interpretations of naturalness, usefulness, and generality. Therefore, the search for a process model which is a conceptual aid to the designer, has universal or near-universal applicability, and can be formalized well enough to serve as the basis for theoretical study, is not over.

Shared Variable Models

Shared variable models of asynchronous interaction arose because programs running under multiprogramming systems communicate efficiently through shared memory locations. The major work on shared variable modeling appears in [9] (the information structure model of [8] is equivalent), and the formulation here resembles Horning and Randell's strongly because they seem to have found the only one which works! Our intention is to capture the essence of all possible shared variable models.

Let us first consider an isolated process P. It has a state space S, the set of all possible states s in which the process can be. A computation of P is a sequence (finite or infinite) of members of S.

If P is to interact asynchronously with another process, then the processes must share some portion of their state spaces. So that this portion can be identified, states must be divided into a fixed number of named components, called variables. Each variable v has an associated value space V containing all the values it can assume. Thus S is the set of all combinations of values of the individual state variables.

A change in the state of P is called a process step, and occurs through assignment of new values to variables. This is formalized as an action relation f, whose domain is S and whose range is the set of all sets of assignments. An assignment is a pair (v,u) where v is the name of a variable and u is a member of its associated value space.

Assuming that we know what it means to apply a set of assignments to a state, then a computation of P is a sequence of states $s_0, s_1, s_2, \ldots s_i, \ldots$ such that $s_i \in S$ $(i \geq 0)$ and s_{i+1} is the result of applying some member of $f(s_i)$ to s_i $(i \geq 0)$.[c] This is shown in Figure 1.

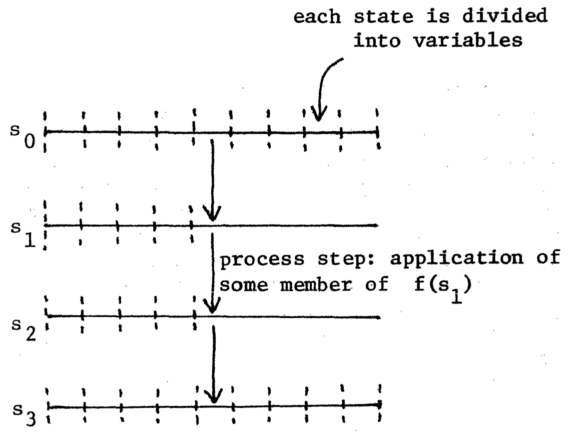

each state is divided into variables
s_0
s_1
process step: application of some member of $f(s_1)$
s_2
s_3

Figure 1. A computation of an isolated process.

As long as a value of $f(s_i)$ contains no two assignments with the same first element, its application is straightforward: all variables v with no assignments (v,u) in the set retain their former values in s_{i+1}, and all variables v with assignments (v,u) in the set take on the values u in s_{i+1}.

The case of two assignments to the same variable is a race condition, or conflict. It would do absolutely no good to define it away at this stage, because it will reappear when we compose processes, anyhow. To deal with it, we have exactly two choices: to make the resultant value of the variable one of the assigned values nondeterministically, or to make it undefined. Rather than letting computations be stopped dead by undefined states, we will choose the former. Thus a process is deterministic, meaning that only one computation can be generated from a given initial state, only if (a) the action relation is a function, and (b) no value of the action function contains more than one assignment to the same variable.

Since there is no mechanism through which a process P_1 can interact with external entities, the only way to study the interaction of P_1 and another process P_2 is to compose them, forming another process. This can only be done if their state spaces, S_1 and S_2, have some variables

[b] A formalization problem which this paper ignores is that of programming language semantics. Wegner defined a formal process model called an information structure model ([8]) for the purpose of defining and proving assertions about language semantics. The emphasis here is on how a model handles asynchronous interaction, since the need for a dynamic model of computation has only arisen with logical concurrency.

[c] The precise reason why the more complex action function, rather than a successor function (whose range as well as domain is S), must be used cannot be made clear yet. See footnote (d).

in common. For simplicity, it can be assumed that shared variables have the same names in all sharing processes, and that all other variable names are unique. Then the state space of the composite process consists of all variables of S_1 and S_2 with distinct names (in other words, everything in S_1 and S_2 except for duplicates of variables).

Since the action relations of P_1 and P_2 are indivisible, the notion that P_1 and P_2 are proceeding in asynchronous parallel translates to the statement that their relative rates are unconstrained. In other words, each step of the composite process should be interpretable as a step of P_1, a step of P_2, or a step of both (otherwise true parallelism would be excluded), and steps of each type can be arbitrarily interleaved. The three types are shown in Figure 2. Clearly the composition cannot be deterministic.

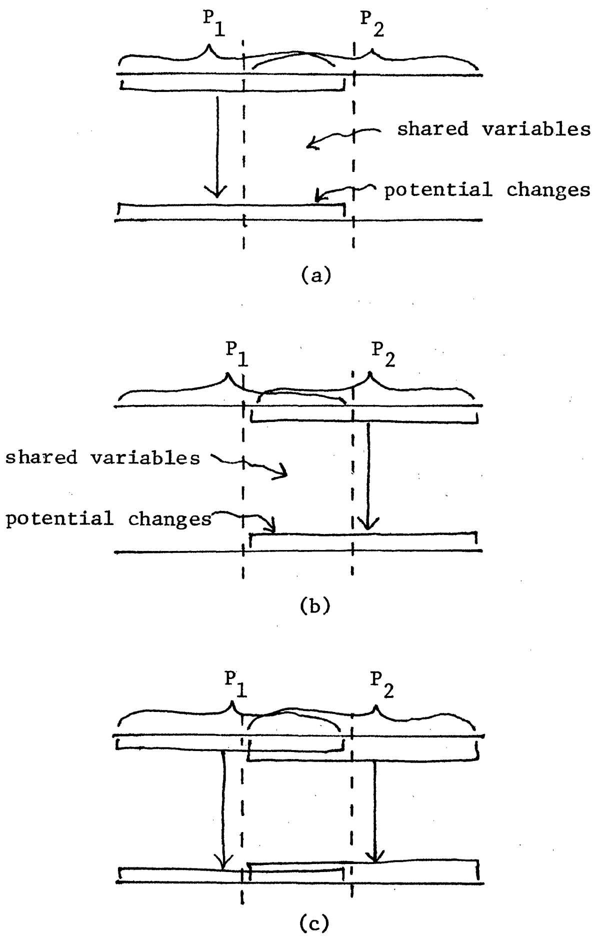


Figure 2. Possible steps of a composite process.

Even if P_1 and P_2 are both deterministic, race conditions can arise when P_1 and P_2 are both active in a step of the composite (Figure 2(c)), and each tries to assign a different value to the same shared variable. As there is still no basis for deciding who won the race, the problem has already been dealt with as satisfactorily as possible in the single-process case. Thus a value of the action relation f of the composite process, with argument s_i of the composite process, is eithei (a) a member A_1 of f_1 (s_{i1}), where A_1 is a set of assignments, f_1 is the action relation of P_1, and s_{i1} is the portion of the state s_i of P common to the state of P_1, (b) a member A_2 of $f_2(s_{i2})$, defined as above on P_2, or (c) $\{A_1:A_1 \in f_1(s_{i1})\}X\{A_2:A_2 \in f_2(s_{i2})\}$, where (A_1,A_2) is interpreted to be the same as $A_1 \cup A_2$. This definition implements the intention described in Figure 2, whether P_1 and P_2 are deterministic or not.(d)

This basic model is easily extended to composition of several processes, initialization, etc. The reader who would like a more thorough explanation of shared variable modeling is encouraged to read [9].

The criterion of naturalness is violated by this model because race conditions cannot be resolved through indivisible operations on shared variables, as is commonly done in real situations. To see why, let us try to model a test-and-set instruction. It will take one step of the executing process, and set a private flag variable to "go" if successful. Now if two processes execute test-and-sets in parallel on the same (unlocked) variable, both will perceive it as unlocked and set their flags to "go". This is not even a case involving conflict on the value of a shared variable, which might cause the state of the composite process to become undefined or doubly defined, because both sub-processes will set the value of the shared variable to "locked."

(d)
The shared variable model has been described informally because its formalization is very messy and adds no insight. This footnote is an attempt to explain why there is no simpler formulation.

The formalization of a process step cannot be factored into separate pieces, one giving the effect of the process step on each variable. The reason is that in a composite process there is no way of telling which variable originated with each sub-process, and thus no way of saying that certain partial relations must always be applied at the same time as other partial relations - yet unless this could be done, composite processes would produce nonsense.

Given that the process step must be formalized as a single relation, the simple successor relation, whose value is the set of possible next states, fails because one cannot separate shared variables for special treatment when the process is composed with another.

In practice, test-and-set instructions, semaphores, etc. work because of the hardware conflict resolution which prevents true parallelism at that level. But since a formal model does not distinguish between hardware and software levels, it is not possible to remove some parallelism without removing parallelism altogether.

There is, of course, a way to enforce mutual exclusion on processes in this model: each competing process communicates with a central arbitrating process through its own shared variable. The problems above disappear because competing processes do not share variables with each other, and so all use of shared variables occurs within a cooperative protocol. This solution does not seem to mitigate the fact that a major method of synchronization in real processes cannot be modeled. Furthermore, since the private shared variables resemble message buffers, it may be fair to say that this is a simple implementation of the concept of message transmission. The implications will be discussed further in Section 4.

The criterion of usefulness is violated by this model because shared variable descriptions do not correspond to reality unless the process step is limited to what can be accomplished in a single machine instruction. This is because steps of parallel sub-processes must begin and end at the same points in time. This situation can only be achieved in an implementation if (1) a sub-process waits until all its fellows have finished their steps before it starts a new one, which is ridiculous; (2) the steps of all sub-processes always happen to finish at the same time, which is equally ridiculous; or (3) each step is a single machine instruction,[(e)] which makes the model no more useful than an assembler listing.

Ideally the action relation of a process would represent its internal computation (during which the process need not be in a well-defined state), no matter how complex it is. Between steps the state is well-defined, and the process absorbs and emits information. This allows delayed binding of the exact form of the internal computation and, consequently, hierarchical design.

Since the process is asynchronous with respect to other processes, its steps do not have to be synchronized with theirs. This is the situation shown in Figure 3(a), and known to every designer of real concurrent processes. It cannot be simulated by the trick in 3(b) because that would force sub-processes to produce well-defined states at arbitrary points in the middle of their steps.

(e) And instruction execution on all sub-processes is synchronized. This will be discussed next.

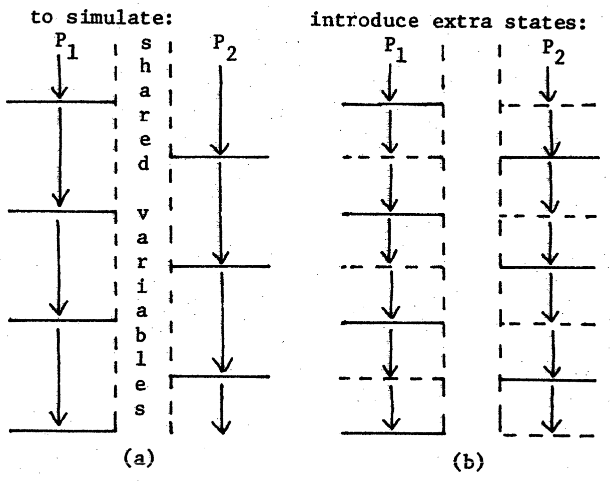


Figure 3. An invalid solution to the process step problem.

Finally, the criterion of generality is violated by this model because it does not include processes running in parallel on different nodes of a network. This is obvious from the preceding argument, in which it was deduced that initiation and completion of process steps on all sub-processes had to be synchronized.

Another argument is that inter-node transmission delays are not modeled. We could, for instance, introduce a shared variable as a model of a one-way communication link between two processes. But as soon as one process writes in it, the information is available to the other process, which is not at all accurate. This structure is also susceptible to a criticism made before: Is it not a simulation of something else, i.e. message transmission?

To summarize the reasons why shared variables cannot be formalized successfully: to define a system of asynchronously interacting processes communicating through shared variables, it is necessary to make a stronger constraint on them than is desirable or possible in practice - that at any discrete point in time at which <u>any</u> state information is well-defined and observable, <u>all</u> state information must be well-defined and observable. Interaction between processes which can be formally modeled with shared variables is synchronized to some degree.

The Message Model

As an alternative, the process model defined in detail in [10] permits asynchronous communication only through message transmission with arbitrary (finite, non-zero) delay. It is deterministic, and intended to include the largest possible class of digital phenomena for which deterministic modeling is feasible. Message modeling does not suffer from any of the problems discussed above because asynchronous sub-processes have disjoint state spaces.

A sub-process which is internally synchronous, but interacts only asynchronously with its environment, is defined as an atomic process. An atomic process has a state space (of arbitrary structure), an initial state which is a member of the state space, and a total successor function. An argument of the function consists of the present state of the atomic process, plus the queue of messages (also of arbitrary structure) it has received since the last time it took a step. A value of the function consists of the next state of the atomic process, plus a vector of messages, each to be sent on one of the asynchronous source-to-destination message paths of which it is the source. The cycle of an atomic process is illustrated in Figure 4.

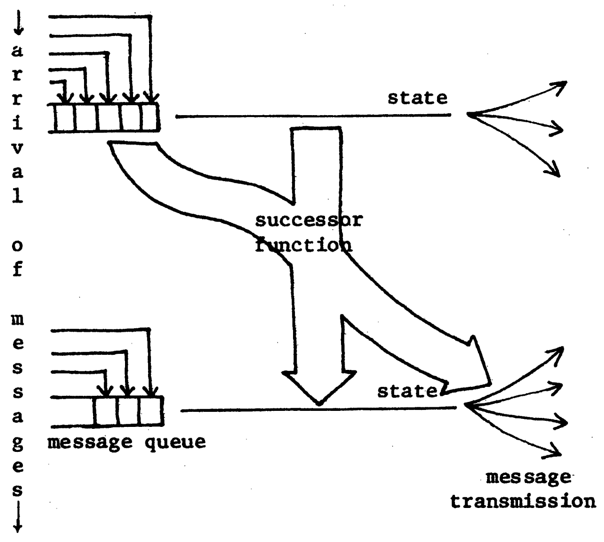


Figure 4. Cycle of an atomic process.

Any atomic or external process can be a message source or destination. An external process is a component with the same behavioral capabilities as an atomic process, but it is outside the process being modeled.

The sending and receiving of messages is modeled as follows. Between any two atomic or external processes there is at most one source-to-destination path in each direction. Each time the source atomic or external process takes a step it can send at most one message on this path, which is received at the destination some arbitrary positive finite time later. Messages do not pass each other on a source-to-destination path. At the destination sub-process, all messages received are saved in a single queue, in arrival order.

In an atomic process, the actual destinations of the independent output streams are still unspecified. A process consists of one or more atomic processes plus a function which specifies these destinations. The separation between output streams and their destinations, and the symmetry between atomic and external processes, provide great flexibility in composing and decomposing processes. A possible process configuration is shown in Figure 5.

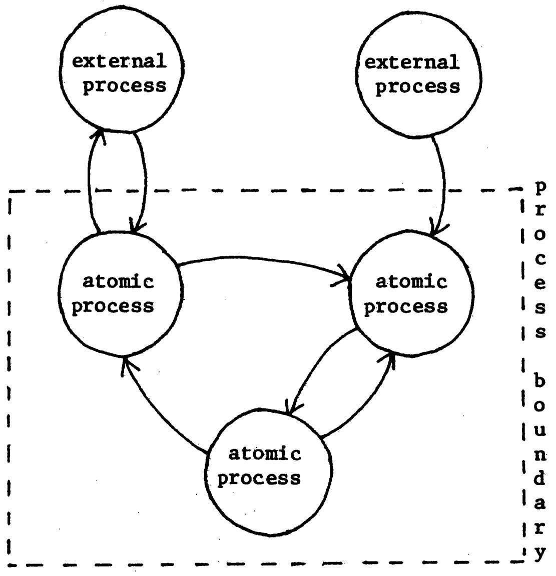


Figure 5. A process configuration.

In [10] it is explained how the behavior of external processes is specified, how the relative rates of atomic processes, external processes, and transmission delays are specified, and how such a system is simulated deterministically. Examples are given indicating that this model satisfies the criteria of naturalness, usefulness, and generality.

Example: Sharing a Data Base

In this section we consider a familiar problem: two concurrent "user" processes must share a data base under conditions of mutual exclusion. In such a situation, shared variable modeling may seem the obvious choice (and may be theoretically sound, if the concurrency is simulated by multiprogramming and the modeling is at the machine instruction level). By considering both models, we will show why the message model seems to be a more powerful conceptual tool, and where the ubiquitous shared variable program structures do fit in.

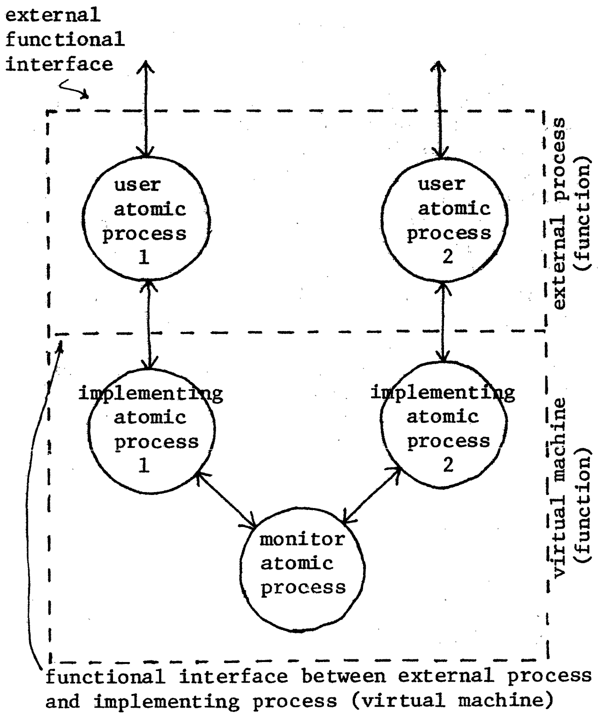
In the shared variable view, the two processes share the space in which the data is stored, and use some cooperative protocol, based ultimately on hardware mutual exclusion, to ensure logical mutual exclusion. If each process passes control to canned programs to perform its part in the protocol, then we have a synchronization primitive such as a semaphore or critical region. If each process passes control to canned programs to perform standard operations on the data, then we have data abstraction ([11]). The combination of the two produces a monitor.

This is indeed a practical structure, but the message model also includes it. It is only necessary to see that a monitor defines an atomic process(f) whose state space includes all the "shared data" - but since only the monitor process uses it, it is not really shared at all. The steps of this process consist of the execution of a monitor procedure between initiation or awakening of a call, and termination or blocking of a call. The monitor process enforces mutual exclusion on the data base by servicing only one access request during each of its steps.

What distinguishes this process structure is the fact that the monitor process is never active concurrently with the calling user process - its logical subordination is echoed in the implementation strategy of passing the physical processor along with the call. This means that the monitor atomic process could not belong to the same process as the user atomic processes, because it is in no sense parallel to them. Where does the monitor fit in?

In [10] it is shown that any process can be identified as a realization of a function (and certainly any computable function can be realized by a process). The data base and its monitor are employed by a user process as it goes about its business of computation. We can therefore conclude that the monitor is an atomic process in the process which realizes the successor functions of the user processes. Thus the users share the monitor process, not the data base, and new light is shed on the statement: "The main difference between processes and monitors is the way they are scheduled for execution" ([12]).

A call on a monitor procedure is a message sent to the monitor atomic process by another atomic process in the lower level (implementing) process. It contains a request for service and parametric information. This structure is illustrated in Figure 6.



external functional interface
user atomic process 1
user atomic process 2
external process (function)
implementing atomic process 1
implementing atomic process 2
monitor atomic process
virtual machine (function)
functional interface between external process and implementing process (virtual machine)

Figure 6. Modeling of a monitor process.

It is the efficient mapping of this logical structure onto a physical realization which produces the familiar constructions. Once an atomic process has sent a message downward in Figure 6, it will enter a passive waiting state until an answer is sent upward. Therefore allocation of the physical processor follows the messages. Since a user process and its corresponding implementing process are really highly synchronized, message transmission between them can be implemented in the degenerate form of shared space, i.e. the state space of the user process. As for messages from the implementing processes to the monitor, only the service request need be queued physically. The parameters, which are logically part of the message, are passed by reference.

In addition to this structure, the message model offers another: The data base process can be truly concurrent with the user processes, as might be expected if they ran on different nodes of a network. Even at a site where all processors (real or virtual) share the same memory, there may be reasons why this is a good strategy. If the processors are real, as in an array of microcomputers, it is a way to distribute the workload among them. If the processors are only virtual performance will not be affected, but the opportunities for explicit cooperation, scheduling, etc. among the users of the data base are increased.

In this case, illustrated in Figure 7, both user processes and the data base process are

(f) Strictly speaking, an atomic process exists only as part of a definition, and becomes a sub-process (not necessarily proper, since a process need contain no more than one atomic process) of a process only when message destinations are furnished. We may use "process", "sub-process," and "atomic process" to refer to the same entity, depending on which usage is most immediately appropriate.

atomic sub-processes of the same process, with no constraints on their relative rates. It would be advantageous to centralize computations on the data in the data base process, both for modularity and maximized parallelism.

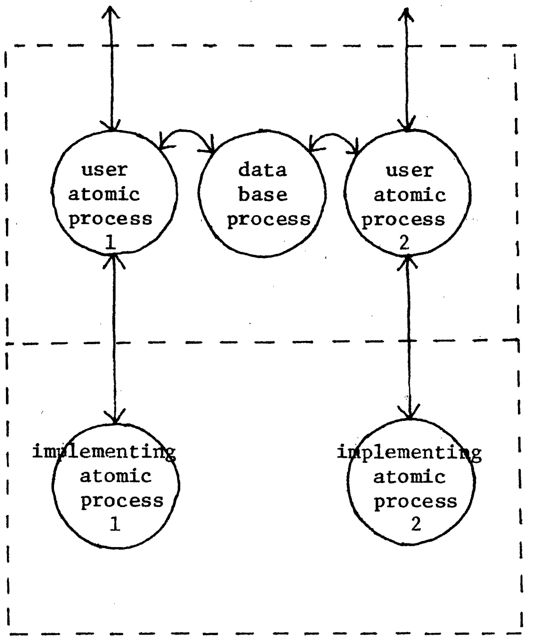


Figure 7. Modeling of a concurrent data base process.

In a multiprogramming or multiprocessing system with shared memory, general message transmission will be simulated by a virtual machine created by the operating system, as it is in the RC 4000 system ([13]). The actual implementation mechanism is likely to be a monitor, as described for the previous alternative.

Thus we can arrive at a tentative characterization of shared variable communication, message communication, and their respective places in process design. It is at least possible to model all mutually asynchronous processes as the sole owners of and operators on their state spaces, communicating with each other through messages. This is a high level, hierarchical concept which can suggest to the designer a variety of structures.

The implementation of these structures is another thing entirely. Between computers, interaction will take the form of data communication. Within a computer, message transmission must be simulated through memory which can be transferred fluidly from the state space of one process to that of the other.

Here it is wise to take advantage of known logical constraints on relative rates and the amount of information which can be transmitted. In many cases it will turn out that a single shared variable is an adequate implementation of message transmission between two processes. Yet it seems that much insight is lost by beginning with the relatively structureless shared variables themselves.

Conclusion

Searching for a process definition that will be equally inspiring to programmers and theoreticians, we find that models based on shared variable communication are formally shaky, and may encourage process designers to limit themselves to a few familiar communication structures.

A model based on message communication is formally sound and apparently applicable to a wider variety of process structures. Yet it remains to be seen whether or not this model will ever have the impact that the shared variable model has had, because earlier work compatible with message transmission has not developed in the direction of influencing the design and implementation of processes. We are encouraged by the writing of [14], in which the message model suggested a high level design tool.

It is the author's belief that differences of opinion about the relative appropriateness of these models have their origin in the unknown territory of degrees of synchronization. When processes are completely asynchronous, as they are on different nodes of a network, there is little doubt that they communicate through messages. Our attempt to formalize the shared variable model has shown that shared variable communication often succeeds because there is some degree of synchronization between the sharing processes, if only the hardware interlock on memory references.

Although we currently take the view that useful degrees of synchronization will be describable as special cases of the message model, more convenient descriptions may also be available. We hope this paper has shown that those more convenient descriptions must be chosen with care.

Acknowledgment

It is a pleasure to acknowledge continuing interaction with D. R. Fitzwater on the subject of process structure.

References

[1] E. W. Dijkstra, "Co-Operating Sequential Processes," Programming Languages, Academic Press, (1968), pp. 43-112.

[2] P. Brinch Hansen, Operating System Principles, Prentice-Hall, (1973), 366 pp.

[3] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," CACM (October, 1974), pp. 549-557.

[4] T. S. Kuhn, The Structure of Scientific Revolutions, University of Chicago Press, (1962), 210 pp.

[5] R. E. Miller, "A Comparison of Some Theoretical Models of Parallel Computation," IEEE Transactions on Computers (August, 1973), pp. 710-717.

[6] J.-L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," Computing Surveys (March, 1973), pp. 31-80.

[7] J. L. Peterson, and T. H. Bredt, "A Comparison of Models of Parallel Computation," Proceedings of the IFIP Congress (August, 1974), pp. 406-410.

[8] P. Wegner, "Operational Semantics of Programming Languages," Proceedings of an ACM Conference on Proving Assertions About Programs (January, 1972), pp. 128-141.

[9] J. J. Horning, and B. Randell, "Process Structuring," Computing Surveys (March, 1973), pp. 5-30.

[10] P. Zave, Functional Equivalence of Parallel Processes, Computer Science Department, University of Maryland, TR-439, (January, 1976), 112 pp.

[11] B. H. Liskov, and S. Zilles, "Programming with abstract data types," Proceedings of an ACM Conference on Very High Level Languages (April, 1974), pp. 50-59.

[12] P. Brinch Hansen, "The Programming Language Concurrent Pascal," Transactions on Software Engineering (June, 1975), pp 199-207.

[13] P. Brinch Hansen, "The Nucleus of an Operating System," CACM (April, 1970), pp. 238-241, 250.

[14] P. Zave, An Approach to Parallel Process Design, Computer Science Department, University of Maryland, TR-452, (April, 1976), 24 pp.

HIERARCHICAL MONITORS

A.M. Lister and P.J. Sayer

Department of Computer Science
University of Queensland,
St. Lucia, Qld. 4067
Australia

Post Office Telecommunications H.Q.
Scottish Mutual House, Lower Brook Street,
Ipswich, Suffolk,
England.

Abstract -- A technique is presented for constructing an operating system as a hierarchical set of monitors. The hierarchy reflects and reinforces the system structure, and extends down to the system nucleus itself. The nucleus is in fact treated as a specialised monitor for handling the central processor. The technique has been used to write a small pilot system for a DEC PDP-15, and experience with the system is reported. The mutual exclusion problem for monitor procedures is discussed, and a viable solution suggested.

Introduction

The concept of a monitor has been developed by Hansen [1] and Hoare [2] as a tool for structuring an operating system. The idea is to control the resources of a computer installation by constructing a scheduler for each class of resource, each scheduler being implemented as a monitor. A monitor may be regarded as a collection of procedures for manipulating a resource, together with any necessary local administrative data. The integrity of data is maintained by making the execution of the procedures of a monitor mutually exclusive in time, and compile-time checks ensure that resources can be accessed only by means of the appropriate monitor. A detailed definition and notation is given in [2], and a particular implementation of monitors is described in [3].

Examples of resource scheduling monitors are a paging system [4], a single resource scheduler, a buffer access controller, and a 'readers and writers' monitor for controlling multiple access to files [2]. These monitors function at the outer levels of an operating system, and rely for their implementation on facilities provided by the system nucleus. As far as we know there have been no proposals to extend monitor functions into the nucleus itself.

In this paper we propose a methodology for operating system construction in which monitors are incorporated at all levels, including the nucleus. It is now generally accepted that operating systems should be constructed in layers, each layer using the facilities of the layer below and providing facilities for the layers above. The advantages [5,6] are essentially those of applying structured programming techniques to non-sequential systems. The basis of our proposal is that the various levels of an operating system can be implemented as a hierarchy of monitors in which the nucleus, or lowest level, is regarded as a specialised monitor for controlling the central processor. The advantages of this approach are

1. unity of structure
2. compile-time protection mechanisms
3. the use of high-level language constructs at all levels, with a consequent increase in programmer productivity and reduction of error rates.

This methodology, which we call hierarchical monitors, is discussed in more detail in the next section. Evidence that the methodology is indeed useful has been derived by constructing a pilot implementation of a small system. The implementation is described in section 3, and the nucleus of the system is compared with that of a similar nucleus constructed entirely in assembly language along 'traditional' lines. Our conclusion is that the overheads involved in the hierarchical monitor structure are small enough to be outweighed by the ease of implementation and the facility of compile-time checking.

Hierarchical Monitors

Two key problems in devising a methodology for operating system construction are

1. how to impose a conceptual hierarchy on the system, i.e. decide which parts of the system should belong to each layer
2. how to map the conceptual hierarchy into an appropriate software structure.

As far as 2. is concerned we hope to show that monitors are indeed suitable units for building the structure.

With respect to 1. a strong case can be made [7] for making the bottom two levels

1. interrupt handler and dispatcher (low level scheduler)
2. interprocess communication mechanisms.

Our proposals retain these two levels in the hierarchical monitor nucleus, where they are implemented as the CPU monitor and IPC (interprocess communication) monitor respectively (see figure 1). Other nucleus functions, such as process creation and deletion, I/O handling, and memory allocation, are implemented as monitors in level 3, while the remaining system functions are in level 4. Level 5 contains user programs. The justification for this structure is as follows.

Firstly, the only occasions on which the dispatcher need be called are

1. after certain interrupts (e.g. time-out)
2. after process completion
3. when a process is blocked or awakened by

use of a synchronisation primitive.
In other words, access to the dispatcher is required by nucleus functions only.

Secondly, access to the interprocess communication primitives wait, signal, and queue [2] must be limited to monitor procedures only, while access to monitor procedures themselves can be allowed at any level.

Hence three classes of privilege emerge:
1. Access to the dispatcher - nucleus monitors only
2. Access to wait, signal, and queue - monitors only
3. Access to other monitor procedures - any level.

In the structure of figure 1 this corresponds to
1. Only levels 2 and 3 can access level 1
2. Only levels 3 and 4 can access level 2
3. Any level can access levels 3 and 4.

Given suitable hardware this pattern of privilege could be enforced by implementing each level as a protection domain, and by including the appropriate capabilities in each domain. However, if each level consists of a set of monitors the following protection can be afforded without use of specialised hardware.

Compile-time protection. Monitor procedures can be called only by quoting both the monitor name and the procedure name. In an implementation of monitors such as that described in [3] these names need not be global to the entire system, but can be restricted in scope to those levels which have legitimate cause to use them. In particular, the names of the IPC monitor procedures (which implement the synchronisation primitives wait, signal, and queue) can be made known only to other monitors, and those of the CPU monitor made known only to nucleus monitors. All data local to monitors, as well as the names of the resources controlled by them, can be protected by compile-time enforceable scope rules.

Run-time protection. The compile-time protection obtained by privacy of names can be supplemented by run-time checks applied on each monitor entry. Each time a monitor procedure is called the monitor can check the privilege level of the caller (held in an element of its process descriptor) before the call is allowed. The monitor thus acts in a manner similar to that of the MULTICS 'gatekeeper' [8] in validating transfers of control from one level of privilege to another. It is worth noting that these run-time checks, which carry an obvious overhead, are necessary only when the implementation of monitors is such that the appropriate privacy of names cannot be guaranteed.

Both forms of protection described above rely on all transfer of control being performed through the legitimate procedure call and return mechanisms. They provide no defence against illegal jumps into the middle of procedure bodies, nor against the construction of illegal data addresses. Such offences can be prevented only by hardware protection mechanisms. However, this should not be seen as a criticism of hierarchical monitors, since the same comment can be made whatever a system's method of construction. The authors' view is that the use of monitors provides a useful first line of protection at compile time, and that this can (and should) be supplemented by suitable hardware protection at run time.

A Pilot System

The hierarchical monitor methodology described in the last section was developed and tested by constructing a small pilot operating system for a DEC PDP-15 computer. The system was written in BCPL [9], using the implementation of monitors described in [3]. In this implementation a monitor is declared as a BCPL procedure, and procedures belonging to a monitor are declared as further procedures within it. Mutual exclusion of monitor procedures is effected by disabling interrupts on monitor entry and re-enabling them on monitor exit. We shall say more about this mechanism in the next section.

A brief description of the monitors found at each level of the system (see figure 1) is given below.

Level 1

This level contains the CPU monitor only. There are two monitor procedures - dispatch and interrupt (n). A call to interrupt acts in a similar way to the traditional extracode or supervisor call, and the parameter n is used to indicate the type of service required. Interrupt is one of the few procedures which need to be partially coded in assembly language (the others are concerned with driving I/O devices). The descent to assembly language is made for saving and restoring machine registers, and also for handling externally generated (hardware) interrupts.

The function of dispatch is to switch the CPU between processes. It saves the current process's environment, chooses the next process to run, and restores the new environment.

Level 2

The IPC monitor is the sole monitor at this level. It contains the synchronisation procedures wait, signal, and queue. These procedures could of course be replaced by others if it were decided to base process synchronisation on a different set of primitives. A listing of the IPC monitor is given in Appendix 1; it is hoped that the notes will make the monitor understandable even by readers not familiar with BCPL. Notice that the Boolean usermode (which is in each process descriptor) is normally used by a monitor to determine whether it is being called from a user program, in which case exclusion has to be gained, or from another monitor, in which case exclusion has already been obtained. In the case of the IPC monitor, which cannot be called directly from a user program, the exclusion should already have been obtained, and so usermode is used as an extra error check. Note also that both wait and signal call the CPU monitor to effect process switching.

Level 3

There are several monitors at this level - the memory monitor, the process monitor, and an I/O monitor for each peripheral device.

The memory monitor contains procedures for allocating and retrieving memory. In the current system allocation is made in arbitrarily sized blocks from a free chain, but in larger systems the allocation algorithms could be expected to reflect the architecture of the machine involved. In a paged system the memory monitor would call an I/O monitor to transfer pages to and from backing store.

The process monitor contains two procedures - one for the creation of processes and the other for deletion. Both procedures call the memory monitor, and deleteprocess calls the CPU monitor to effect process switching.

Input and output is handled by an I/O monitor for each device. A typical monitor contains procedures for initiating transfers, and in the case of a shareable device such as a disc, it might also contain procedures for scheduling access. In the present system only teletype I/O has so far been implemented (a listing of the teletype monitor is given in Appendix 2), but monitors for other devices would follow a similar pattern. The outline of a disc monitor is given in Appendix 3.

The interface between I/O monitor procedures and the interrupt handler should perhaps be elaborated. The busy/ready status of a device is represented by a Boolean variable which is tested inside the monitor before I/O is performed. If the device is busy the requesting process waits on a condition variable associated with the device (see, for example, the teletype monitor in Appendix 2). The interrupt handler signals the condition variable when the device interrupts to say it is ready. Some distortion of the hierarchical structure is necessary here: the interrupt handler cannot be said to be called from the I/O monitor since it is certain that another process will be running when the interrupt occurs. In this specific case the scope rules of moniotrs are bent to allow the interrupt handler (part of the CPU monitor) to signal condition variables declared in an I/O monitor. It happens that this can be easily accomplished, since the interrupt handler and the device driver portion of an I/O monitor are both written in assembly language and hence can be assembled together.

Level 4

This is the level at which other system or user monitors occur. These monitors are concerned with such things as buffer allocation, file access, device allocation, and so on. Several typical monitors have been written for the pilot system: experience with these monitors is described in the next section. Although level 4 at present contains all such monitors there is no intrinsic reason why it could not be split into several separate levels, with the monitors allocated to each level as appropriate. This method of extending protection to the higher level functions might commend itself in large systems.

Level 5

All user (non-monitor) programs are found at this level.

Construction of the pilot system establishes that it is possible to use the hierarchical monitor technique to build a system nucleus. It does not in itself establish that the technique is an improvement on already existing methods. As a control exercise the hierarchical monitor nucleus was compared with a similar nucleus written entirely in Macro-15 assembly language, but using the more traditional P and V operations as synchronising primitives.

The overheads of the hierarchical monitor nucleus as opposed to the assembly language nucleus are

1. the overhead of the high-level language
2. the overhead of the monitor implementation
3. the overhead of imposing the hierarchical structure and error checks.

It was found that in terms of the number of machine code instructions produced the total overhead was about 200%. Of this about half is caused by use of a high-level language: the code produced by our compiler is not very compact, and can be greatly optimised for such specialised cases as the system nucleus. We estimate that about half the remaining overhead is incurred by the way in which monitors are implemented (each monitor call involving two BCPL procedure calls and a case statement). Bearing in mind the possibility of a better compiler and a better implementation of monitors we suggest that the true overhead of the hierarchical monitor nucleus lies between 40% and 60%. Against this overhead should be set the compile-time checking, higher programmer productivity, and increased reliability afforded by the hierarchical monitor technique.

We have already mentioned that part of the interrupt procedure in the CPU monitor is written in assembly language. This section of code, which tests device flags and saves machine registers, is 82 instructions long. The lowest level of I/O, that of initiating transfers, is also coded in assembly language, as is the BCPL run-time system which comprises

1. procedure entry and exit code
2. arithmetic routines
3. subroutines which operate on the run-time stack
4. BCPL features, such as bit selectors
5. BCPL routines, such as packstring, longjump, and lock

which are implementation dependent. Nevertheless, over 90% of the nucleus is written in BCPL. Some change in this proportion might be expected if other forms of I/O were added, but this change is not likely to be large since all I/O monitors need use assembly language only at the lowest level.

The Mutual Exclusion Problem

It will be recalled from the last section that interrupt inhibition is used in the pilot system as the technique for ensuring that the execution of monitor procedures by different processes is mutually exclusive. Although this gives rise to no problems in the pilot system, where the interrupt rate is relatively low, the length of time for which interrupts are disabled might be unacceptable in other, larger, systems. In this section we consider alternative means of ensuring mutual exclusion, and examine whether interrupt inhibition is in fact as dangerous as it appears.

In considering exclusion mechanisms we make a distinction between 'local' and 'global' exclusion. By local exclusion we mean that only the procedures of each separate monitor are mutually exclusive; global exclusion means that all procedures of all monitors are mutually exclusive. Clearly, only local exclusion is necessary to guarantee the integrity of the data and resources administered by each monitor.

Unfortunately, local exclusion is difficult to implement in situations where arbitrarily nested monitor calls are allowed. The difficulty (described more fully in [3]) arises in recording the exclusions acquired by a process which executes a nested sequence of monitor calls, and more particularly, in restoring those exclusions when the process is resumed after a wait operation. Consideration of this difficulty led to the decision to use a global rather than a local exclusion mechanism in the pilot system.

We note in passing that if nested monitor calls are forbidden a suitable implementation of local exclusion is a 'test and set' instruction which acts on a separate memory location for each monitor. An alternative is to use P and V operations on semaphores, but while this has the advantage of avoiding the busy waiting that can be incurred by a test and set instruction it has the serious disadvantage of adding another set of synchronising primitives to the system.

The inhibition of interrupts is of course a (somewhat drastic) global exclusion mechanism. It can be used only when

1. the time spent in executing a monitor procedure does not exceed the crisis time of any interrupting device

and

2. there is only one CPU in the configuration.

We examine these conditions (both of which are satisfied in the pilot system) in turn.

The crucial factor affecting condition 1. is the length of monitor procedures. In the pilot system all the procedures are short, the longest taking 760 microseconds to execute. The pilot system is, however, a small one, and it is worth looking at the kind of monitor which might occur in larger systems. One such monitor, the longest we have come across, is Hoare's paging monitor [4] which implements a virtual memory system. This monitor uses a memory allocator (similar to our memory monitor) to allocate page frames, and a 'drummer' monitor to effect page transfers. The time spent inside the monitor is broken up by a series of wait operations, for memory to become available and for page transfers to be completed. The occurrence of such wait operations is important, since a process which executes them releases exclusion. It appears that the critical factor is not the length of the monitor itself, but the length of code between successive wait operations. In most situations we would not expect this to be more than a few instructions.

Another point worth making is that when monitors are used for all system functions exclusion may be granted even when it is not needed. For example, in the pilot system the procedure createprocess (part of the process monitor) creates and initialises a process descriptor and then chains the descriptor into a global data structure. Exclusion is required only for the last of these operations, but it is granted for all of them. There may be a case here for allowing explicit release and acquisition of exclusion inside monitor procedures, but this would imply the possibility of a whole range of programming errors which monitors are specifically designed to prevent. It would seem more appropriate to revise design ideas so that only those system functions which require some form of exclusion are implemented as monitor procedures.

With regard to condition 2. above, a global exclusion mechanism for a multi-processor configuration can be provided by means of a test and set instruction which acts on a single memory location for all monitors. Such a mechanism effectively replaces the crisis time problem by a busy waiting problem. The proportion of processor time spent in busy waiting will be dependent on the time spent in executing monitor procedures, and the earlier comments on this topic still apply.

To summarise, our experience indicates that local exclusion mechanisms are ruled out by problems of implementing nested monitor calls, but that global mechanisms such as disablement of interrupts or test and set are viable in nearly all situations.

Conclusion

In this paper we have suggested hierarchical monitors as a design methodology for operating systems. We have shown that such a methodology reflects and reinforces the structure of an operating system, and that it can be applied to all levels of the system, including the nucleus. The pilot system indicates, within the limits of any small scale model, that a full size operating system could be constructed along these lines. Further work is necessary, however, to confirm that the methods of this small scale study are appropriate for large commercial systems.

References

[1] P.B. Hansen, "Structured multiprogramming," Comm. ACM (July, 1972), pp.574-578.

[2] C.A.R. Hoare, "Monitors: an operating system structuring concept," Comm. ACM (October, 1974), pp.549-557.

[3] A.M. Lister, and K.J. Maynard, "An implementation of monitors," Software Practice and Experience (July, 1976), pp.377-385

[4] C.A.R. Hoare, "A structured paging system," Computer J. (August 1973), pp.209-215.

[5] A.M. Lister, Fundamentals of Operating Systems, Macmillan, (1975), 144 pp.

[6] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured Programming, Academic Press, (]972), 220 pp.

[7] E.W. Dijkstra, "Hierarchical ordering of sequential processes." In Operating Systems Techniques, ed. C.A.R. Hoare and R.H. Perrott, Academic Press, (]972), 390 pp.

[8] R.M. Graham, "Protection in an information processing utility," Comm ACM (May, 1968), pp. 365-369.

[9] M. Richards, "BCPL - a tool for compiler writing and systems programming," AFIPS Conference Proceedings SJCC, (1969), pp. 557-566.

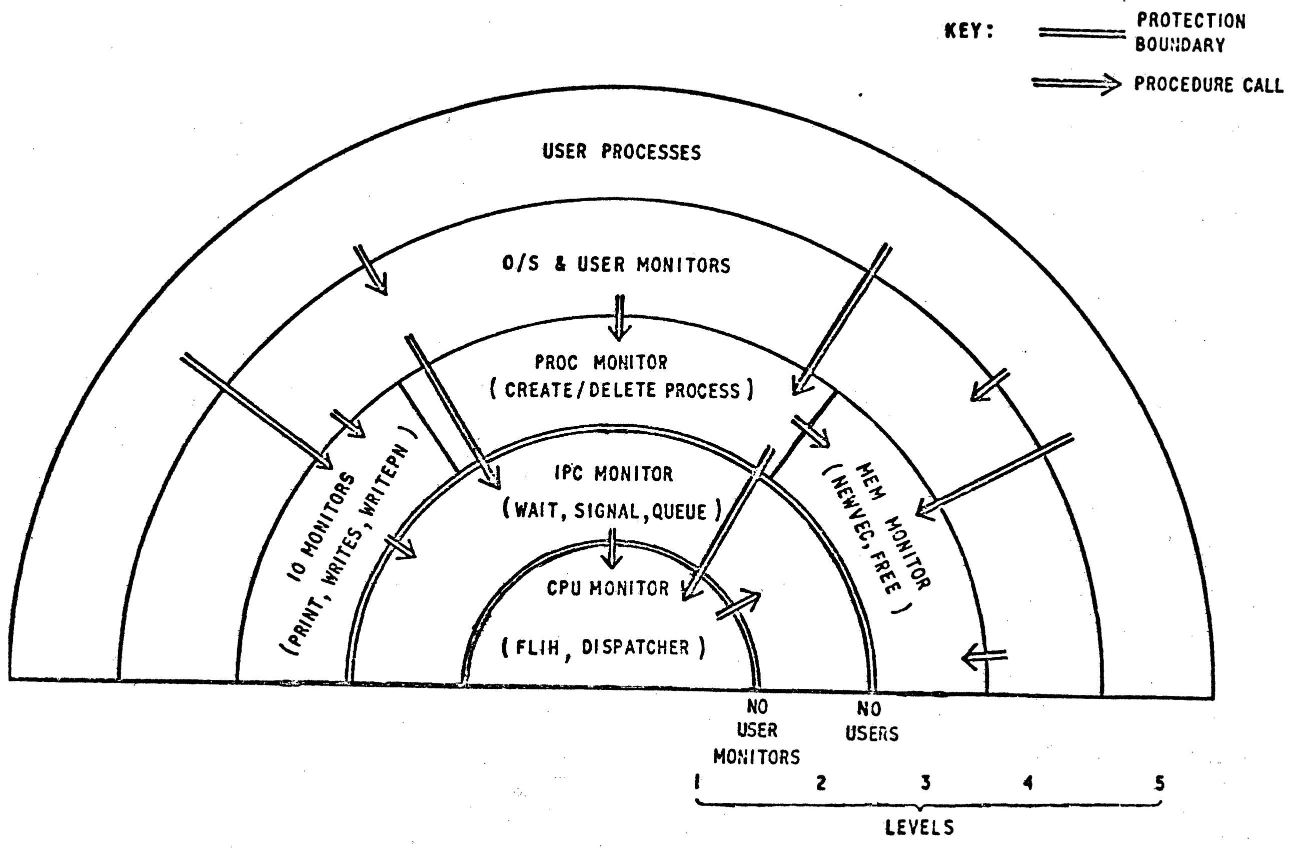


FIG.1 IMPLEMENTATION OF THE HIERARCHICAL MONITORS NUCLEUS

Appendix 1

Listing of the IPC monitor

```
let IPC.MONITOR (procedure, condvar) = valof
//parameters indicate which monitor procedure and
//which condition variable;
$(
//first come declarations of monitor procedures;
  let  wait.(condvar) be
  //note 3;
  // wait operation: add calling process to queue
  // on condition variable;
  $( test condvar ! head = 0
  // queue is empty (notes 1,2);
     then
     $( condvar ! head := curproc;
        condvar ! tail := curproc
     // process descriptor forms new queue;
     $)
     or
     $( (condvar ! tail) ! cond.var.ptr := curproc;
     //note 4;
         condvar ! tail := curproc
     // add process descriptor to queue;
     $);
     runnable ! curproc := false;
     // set process not runnable;
     CPU.MONITOR (dispatch)
     // switch to next process;
  $)
  and signal. (condvar) be
  // signal operation: free first process, if any,
  // on queue;
  $( if condvar ! head = 0 then return;
  // queue is empty: do nothing;
     nextproc := condvar ! head;
  // take first process on queue;
     test condvar ! head = condvar ! tail
  // only one process on queue;
     then condvar ! head := 0
  // now no process on queue;
     or condvar ! head := (condvar ! head) !
       cond.var.ptr;
  // or simply detach process;
     runnable ! nextproc := true;
  // set process runnable;
     priority ! nextproc := max priority;
  // force dispatcher to choose it;
     if flih.ptr ! sigptr then return;
  // signal was called from interrupt handler;
     CPU.MONITOR (dispatch)
  // switch to freed process;
  $)
  and queue. (condvar)
 // queue operation: gives true if queue not empty;
   = (condvar ! head = 0 → false, true);
 // end of procedure declarations:
 // monitor body starts here;
 let res = 0;
 // used to pass out result of queue operation;
 if flih.ptr ! sigptr then
 // monitor was called from interrupt handler;
   $( signal.(condvar); return $);
 // so return;
 if curproc ! usermode then error action;
 // monitor cannot be called from user level;
switchon procedure into
//enter appropriate monitor procedure;
$( case wait: wait.(condvar);endcase
   case signal:signal.(condvar):endcase
   case queue: res := queue.(condvar);endcase
   default: error action
$);
resultis res
$)
```

Notes

(1) A condition variable is represented as a vector of two elements, which point to the head and tail of the associated queue.
(2) '!' is the BCPL selector for elements of a vector.
(3) '.' is a valid character in a BCPL identifier.
(4) cond.var.ptr is the offset in a process descriptor of the element used to chain the descriptor into a condition variable queue.

Appendix 2

Partial listing of the teletype output monitor.

```
 let  TTYOUT.MONITOR(procedure,P1) be
 // P1 is used as parameter of whatever monitor
 // procedure is called;
 $(
   // declarations of local variables;
   static  $( tty.busy = false;
              // teletype busy flag.
              tty.condvar = 0
              // condition variable;
           $);
   let key = false;
              // used to retain exclusion;
   // declarations of monitor procedures;
   let writech.(char) be
   // outputs single character char;
   $( if tty.busy then
     IPC.MONITOR(wait,tty.condvar);
   //note 1;
     machine code segment to output character
   $)
   and writepn.(n) be
   //note 3;
   // outputs positive integer n recursively;
   $(if n > 9 then writepn.(n/10);
     writech.(n rem 10 + '0')
   $)
   and remaining procedure declarations be
         .
         .
         .
   // start of monitor body;
   if curproc ! usermode then
   // exclusion not previously obtained;
   $( lock ( );
   // obtain exclusion (note 2);
     key := true;
     curproc ! usermode := false
   $)
   switchon procedure into
   // enter appropriate monitor procedure
   $( case writech : writech.(P1); endcase
    case writepn : writepn.(P1); endcase
          .
          .
          .
    default : error action
  $);
  if key then
  // exclusion must be released;
  $( curproc ! usermode := true;
     unlock( );
  // release exclusion (note 2);
  $)
$)
```

Notes

(1) tty.condvar will be signalled by the teletype interrupt handler.
(2) lock is a procedure which disables interrupts; unlock re-enables them.
(3) to avoid redundancy it would perhaps be better for procedures such as writepn to be incorporated in a library, and to call the appropriate monitor procedure (in this case TTYOUT.MONITOR(writech,..)) to effect the actual output.

Appendix 3

Outline of a disc control monitor.

```
let DISC.MONITOR (direction, block,
    memory.address) be
// performs disc transfer between block and
// memory.address in specified direction;
$(
   // declaration of local variables;
   static disc.condvar = 0;
   // a condition variable;
   let cylinder = 0
   // used for cylinder number;
   let key = false;
   // used to retain exclusion;
   let HEAD.MONITOR (action, cylinder) be
   $( A monitor which schedules the disc head so
      as to optimise head movement.  A particular
      example is given by Hoare [2].  The first
      parameter indicates whether the head is to
      be acquired or released, the second specifies
      the cylinder for the desired transfer.
   $)
// declaration of DISC.MONITOR procedures;
and read.(block, memory.address) be
$(  cylinder := some function (block);
    // calculate cylinder number;
    HEAD.MONITOR(request, cylinder);
    // acquire head;
    initiate transfer from block
      to memory address;
    IPC.MONITOR(wait, disc.condvar);
    // wait for transfer complete;
    HEAD.MONITOR(release, cylinder);
    // release head;
$
and write.(block,memory.address) be
$   .
    .
    .
$)
// monitor body starts here;
if curproc ! usermode then
// exclusion not previously obtained;
$(  lock( );
    key := true;
    curproc ! usermode := false
$)
switchon direction into
$( case read :
         read.(block, memory.address); endcase
   case write :
         write.(block, memory.address);endcase
   default:  error action
$
if key then
// exclusion must be released;
$( curproc ! usermode := true;
   unlock( )
 $)
$)
```

GARBAGE COLLECTION WITH MULTIPLE PROCESSES: AN EXERCISE IN PARALLELISM

Leslie Lamport
Massachusetts Computer Associates, Inc.
26 Princess Street
Wakefield, Massachusetts 01880

Abstract

Dijkstra et. al. have described an algorithm which allows a garbage collector process to run concurrently with a list processing process. Very little overhead is added to the list processor. We show that this solution will work for multiple list processes if they obey a simple restriction on how they are synchronized. We also show how the garbage collection can be speeded up by the use of multiple processes. This is done by parallelizing the sequential collection algorithm.

Introduction

Dijkstra et. al. [1] have described a garbage collection algorithm in which a list processing _mutator_ process runs concurrently with a _garbage collector_ process. This algorithm suggests two further problems: (1) to make the algorithm work if there are several concurrently executing mutator processes, and (2) to speed up the garbage collector by using multiple processes. In this paper, we solve these problems by (1) introducing some constraints on how the mutators may cooperate, and (2) "parallelizing" the sequential collection algorithm. As multiprocessor computers become available, techniques for the parallel execution of algorithms will become increasingly important. We hope that our solution to this parallel processing problem will contribute to the development of these techniques.

The primary goal of the original solution was to keep the mutator's overhead to a minimum, so no unnecessary synchronization of the mutator and the collector was introduced. (This is in contrast to the usual approach to such a problem, as in [2].) We shall maintain this goal in our solution. However, we will introduce some extra synchronization overhead into the collecting algorithm when it is executed by multiple processes. Experience indicates that we usually cannot expect to find an n process algorithm which runs n times as fast as a sequential one, even when computational complexity arguments indicate that it is possible.

The history of the algorithm in [1] shows that the concurrent garbage collection problem is remarkably difficult. Incorrect solutions with false correctness proofs survived the scrutiny of the authors for surprisingly long times, and one was actually submitted for publication before its error was discovered. (Only the existence of a careful, formal correctness proof gives us confidence in the final solution.) Given this history, it would seem almost hopeless to look for a completely new solution for multiple mutators and collectors. We will therefore start with the two process solution, and will modify it for execution by multiple processes in such a way that the correctness of the original algorithm is maintained.

This is an exercise in the parallel execution of sequential algorithms, and we will employ standard parallelizing techniques. However, unlike previous methods which find the parallelism using only the sequential program, we will use our knowledge of why the program is correct in order to transform it for parallel execution. We do not yet know how this method can be generalized beyond this one problem. However, most multiprocessing problems are difficult, especially when unnecessary synchronization must be avoided. Starting with a more sequential algorithm should simplify the problem in many cases.

The Original Algorithm

We begin with a brief sketch of the original algorithm which will enable the reader to follow the rest of the paper. However, for a complete understanding he must read [1]. We assume a data structure consisting of a directed graph composed of nodes and edges. A node has two edges, each of which either points to a successor node or else is null. Certain fixed nodes are designated as _roots_, and a node is said to be _reachable_ if there is a path to it from a root. A nonreachable node is called a _garbage_ node. A process can perform the following two separate, indivisible operations: (1) find the destination of a given edge, and (2) change the destination of an edge.

We assume that the mutator will never make an edge point to a garbage node. However, by changing edges the mutator can turn reachable nodes into garbage. The collector must identify the garbage nodes and change them back into reachable ones by adding them to a particular part of the graph called the _free list_.

To solve this problem, we let each node have a _color_ which may be either white, grey or black. All nodes are initially white. A node is said to be _shaded_ if it is grey or black. We introduce the following indivisible operations: (1) change a node's color to a specified value, and (2) shade a node. The latter operation makes a white node grey and leaves a grey or black node unchanged.

The algorithm requires that after changing an edge, the mutator must then shade that edge's new target node. By properly encoding the colors, this shading operation can be done by just setting a bit. The only other mutator overhead is the synchronization needed because it can try to remove a node from the free list while the collector is adding a node to it. This need be no more costly than the usual overhead of testing if the free list is empty before trying to remove a node from it.

The collector repeatly cycles through the following algorithm, where N denotes the set of all nodes and ϕ is the empty set.

```
marking phase:
   shade all roots ;
   S := ∅ ;
   while S ≠ N
      do choose n ∈ N ;
         S := S ∪ {n} ;
         if n is grey then shade each successor of n ;
                           color n black ;
                           S := ∅
         fi
      od ;
collecting phase:
   for all n ∈ N
      do if n is white then put n on the
                                free list
                       else color n white
         fi
      od
```

This is an equivalent but slightly different version of the algorithm from the one described in [1]. The choice of n in the marking phase is arbitrary, but choosing a node in S obviously accomplishes nothing. The marking phase will eventually terminate if a node n not in S is always eventually chosen.

The correctness of the algorithm is deduced by proving that during the marking phase, the following two assertions are true.

<u>P2</u>. All roots are shaded, and for each white reachable node there exists a "propagation path" leading to if from a grey node, which consists solely of edges with white targets.

<u>P3</u>. No edge can point from a black to a white node unless the mutator has just changed it and has not yet shaded its destination.

P2 is the crucial property. It implies that after the marking phase terminates, there are no grey nodes and all white nodes are garbage. However, proving the invariance of P2 required proving the invariance of the stronger assertion P2 <u>and</u> P3.

Multiple Mutators

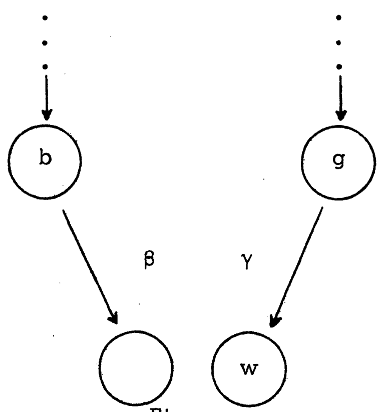
We now consider the problem of allowing multiple mutators to use the list structure concurrently. The mutators must obviously be synchronized in some way so they do not interfere with one another. E.g., if a mutator reads the destination of an edge and then performs some operations which require that the edge retain that destination, then synchronization is required to prevent another mutator from changing that edge. We will not concern ourselves with the implementation of this synchronization, since it will depend upon the details of the individual application.

In the correctness proof of the original algorithm, the only condition to be verified for the mutator is that it leaves P2 <u>and</u> P3 invariant. For multiple mutators, we have the following obvious generalization of P3.

<u>P3'</u>. No edge can point from a black to a white node unless some mutator has just changed it and has not yet shaded its destination.

The proof that the marking phase leaves P2 <u>and</u> P3' invariant is the same as the proof that it leaves P2 <u>and</u> P3 invariant. To prove the correctness of the collection algorithm for multiple mutators, we need only show that the mutators leave P2 <u>and</u> P3' invariant.

It is easy to see that the mutators leave P3' invariant. Hence, we must only show that they leave P2 invariant. Unfortunately, without some further assumption, they do not. For example, assume that the list structure is initially as shown in the Figure, where b is black, g is grey and w is white. Suppose that two mutators then



<u>Figure</u>

perform the following actions in the indicated sequence.

<u>Mutator A</u>	<u>Mutator B</u>
make β point to w	
	make γ point to b
	shade b
shade w	

P2 is false from the time mutator B changes the edge γ until mutator A shades w . If the collector finishes its marking phase during this period, then the collecting phase can incorrectly identify w as garbage and put it on the free list.

In this example, mutators A and B are closely cooperating in their use of the list structure. If B had changed γ before A changed β , then w would have been garbage when A made β point to it. We will place the same restriction on the multiple mutators that we place on a single one: namely, an edge cannot be changed to point to a garbage node. Hence, no "transient garbage" is allowed. This means that A and B must be synchronized so that A changes β before B changes γ . We will require that this synchronization also insure that A shades w before B can change γ .

This requirement is generalized as follows. We assume that the synchronization mechanism enforces some partial ordering $\Rightarrow$ on the mutators' operations, where $e \Rightarrow f$ means that the entire operation e , consisting of changing an edge and shading its destination, is performed before the operation f is begun. We require that if the operations of the mutators were to be performed in any sequential order consistent with this partial ordering - i.e., in any order such that if $e \Rightarrow f$ then e is performed before f - then this is a valid sequence of mutator operations. In other words, the partial ordering must be enough to guarantee that the mutators correctly execute some sequential mutator algorithm.

With this assumption, we now show that the mutators leave P2 invariant. Suppose some mutator operation e makes P2 false. Then e must change an edge and thereby make its former destination n a reachable node with no propagation path to it. This implies that after the operation, every path from a root to n has an edge pointing from a black to a white node. By P3', each of these edges has just been changed by a mutator operation which is not yet completed, and which is thus unordered relative to e by the ordering $\Rightarrow$. Hence, all of these mutator operations could have been performed after e . It is easy to verify that if they were performed after e , then n would be made a garbage node by the operation e and would then be made the destination of an edge by a subsequent operation. This contradicts our assumption that the reordered sequence of operations must be valid, proving that the mutators leave P2 invariant.

The mutators must synchronize their activity when removing nodes from a common free list. Synchronization delays can be reduced by using several separate free lists. The use of multiple free lists presents no correctness problem, and can be implemented without any difficulty. We will not consider it further.

Multiple Collectors

We will parallelize the collection algorithm in three steps: first separately parallelizing each of the two phases, then executing the two phases concurrently. We begin with the marking phase. The node coloring performed by this phase is easily done by separate, concurrently operating processes. We simply divide up the set N of nodes into (not necessarily disjoint) subsets N_i , and have each marker process cycle through the marking loop for the nodes in one of the N_i . For a collector in which finding and shading the successors of n and coloring n black is all one operation, it is obvious that such a parallel execution is equivalent to the sequential algorithm. This is not true for the case of interest, in which examining an edge and shading or coloring a node are separate operations. However, it is easy to check that essentially the same proof given in [1] for the "fine-grained" collector proves that the parallelized version also leaves P2 and P3' invariant. Hence, this is a correct parallelizing of the sequential marking procedure. The only problem is when to terminate the marking phase.

The invariance of P2 implies that the sequential marking phase can be terminated any time after all the nodes have been examined without finding a grey one. (Extra iterations of the loop body do nothing.) However, if a grey node is found, then any node which was previously found to be white might have subsequently become grey. Hence, a new marking cycle must then begin. For the parallel version, this means that when any marking process finds a grey node, it must cause all the markers to restart their cycle. We can thus write the algorithm for the ith of M markers as follows.

```
S := $\emptyset$
while $S_1 \cup \ldots \cup S_M \neq N$
   do while $S \neq N_i$
      do choose $n \varepsilon N_i$ ;
         $S_i := S_i \cup \{n\}$ ;
         if n is grey then shade each successor of n ;
                           color n black ;
                           for j := 1 until M
                              do $S_j := \emptyset$
                              od
         fi
      od
   od
```

If there is some locality condition on the list structure which restricts the set of nodes to which a single edge can point, then one can show that S_j need only be set to $\emptyset$ if n can have a successor in N_j .

This parallel algorithm requires synchronization among all the markers. It is written in a form that suggests an implementation by an array computer. For execution by loosely coupled independent processors, process i would not actually set S_j to $\emptyset$ for $j \neq i$. Instead, it would send some sort of signal to process j . The actual details will depend upon the characteristics of the system with which it is implemented. The interested reader can provide an implementation using his own favorite synchronization mechanism. We will simply assume that there is some way of starting the next phase after the marking phase is finished.

We next consider the collection phase. It is easy to execute this phase with multiple collector processes. The only requirement is that they be synchronized so that two different collectors do not try to change the same edge at the same time. A simple approach is to partition the nodes into disjoint sets, and use a separate process to collect the garbage in each set. Each collector can first "neatly stack" the garbage in its set, and then add the entire "stack" of garbage to the free list by essentially the same operation as adding a single node. Each collector then only performs one short operation which must be synchronized with the other collectors, so the overhead caused by this synchronization will be small. The synchronization required between a collector and a mutator when nodes are added to the free list is the same as for the original algorithm.

We have now parallelized each phase, but the markers are idle while the collectors are running, and vice-versa. To run both phases concurrently, we will pipeline them. I.e., we will perform the (i + 1)st execution of the marking phase concurrently with the ith execution of the collecting phase. This is possible because the collectors collect only already identified garbage, and the markers cannot mark that garbage.

There is one obvious problem with pipelining the two phases. All nodes must be made white before the marking phase is begun if it is to accomplish its purpose; but the collector expects all white nodes to be garbage. There is an obvious solution to this problem. Before executing the two phases, we change all white nodes to some new color, say purple, and color all black nodes white.[a] The collectors will then collect purple nodes and color them black before adding them to the free list. The markers will ignore purple nodes. The mutators can then never make an edge point to a purple node, so it is easy to see that a grey node never points to a purple one, and a purple node is never shaded.

It is obvious that this pipelined algorithm is equivalent to the parallelized two phase solution if the collectors are executed first, and the markers are started after they have finished. To prove that the two algorithms are equivalent in general, we need only show that all the collector operations commute with all the marker operations. It is easy to see that this is the case if a collector only changes an edge of a free list node if it is null, and then must make it point to a black node. This condition is easily met by an algorithm for adding nodes to the free list - e.g., for the one in the Appendix of [1].

Our complete garbage collection algorithm thus consists of cycling through the following steps.

1. Wait until all markers and collectors have stopped.
2. Change all white nodes to purple and all black nodes to white or grey (preferably white).
3. Shade all roots.
4. Start the markers and collectors.

Note that in step 2 we have allowed the possibility that a black node is made grey instead of white. This may happen because of concurrent mutator activity. To insure that garbage is eventually collected, we need only guarantee that a node which is not shaded by the mutator will eventually be made white in step 2.

This pipelined algorithm correctly implements the two phase collection algorithm. I.e., after completing step 1, the state of the list structure will be the same as after the end of the marking phase and before the beginning of the collecting phase in some possible execution of the two phase algorithm.

For our multiprocess algorithm to be efficient, steps 2 and 3 must be fast, since all the markers and collectors are then idle. Step 3 will ordinarily be fast, because there should be relatively few root nodes.[b] We must only make step 2 fast. The easiest way to do this is as follows. We define three different hues numbered 0, 1 and 2. Each node has a hue and a grey value, the latter either 0 or $\frac{1}{2}$. The color of a node is the sum of its hue and its grey value. The meaning of the colors is determined by a global variable base, as follows (arithmetic is modulo 3):

$\underline{\text{base}} - 1$ = purple (garbage)

$\underline{\text{base}} - \frac{1}{2}$ = impossible

$\underline{\text{base}}$ = white

$\underline{\text{base}} + \frac{1}{2}$ = grey

$\underline{\text{base}} + 1$ = base + $1\frac{1}{2}$ = black

The mutators' shading operation is done by setting the grey value to $\frac{1}{2}$, so the setting of base need not be synchronized with the mutators. Step 2 is implemented by simply incrementing base by one modulo 3. Since base is changed only in step 2, a marker or collector need only read its value once when it is first started in step 4.

To insure that all garbage is eventually collected, step 2 must not make a node grey unless it was recently shaded by a mutator. This is achieved by simply having a marker reset the grey value to zero when it makes a grey node black. However, the identification of garbage can be speeded up if the markers and/or the collectors reset the grey value for any black node they encounter.

The redundancy in the above encoding implies that it should be possible to save space by using only two hues. Making use of the fact that a marker only blackens an already grey node, we can employ the following encoding (arithmetic is modulo 2):

$\underline{\text{base}}$ = white

$\underline{\text{base}} + \frac{1}{2}$ = grey

$\underline{\text{base}} + 1$ = purple (garbage)

$\underline{\text{base}} + 1\frac{1}{2}$ = black .

However, to make step 2 fast, the grey value of

[a]When the algorithm is started for the first time, the creation of purple nodes must be suppressed.

[b]We can also eliminate step 3 entirely by defining a root node to be permanently shaded.

a node must depend upon a global grey.base.flag. Step 2 complements this flag and increments base by one modulo 2. Unfortunately, this requires synchronization between the mutators and the collector. In particular, step 3 cannot be executed until the completion of any mutator's shading operation begun before grey.base.flag was complemented. This adds extra steps to the mutator operation, but still does not require a mutator to wait for the collector (unless the free list is empty). The details are non-trivial, but will be omitted.

Concluding Remarks

Let us now review the method we used to obtain our solution, and see what general observations we can draw from it. First of all, we observe that instead of starting with precisely the algorithm described in [1], we rewrote it in a somewhat more general form. We allowed the marking phase to examine nodes in an arbitrary order, and even allowed it to do useless operations by choosing n in S . This simplified our proof that the parallelized version was equivalent to the sequential one. We even had to make use of the fact that the body of the marking phase's while loop could be executed after the while condition became false, although that is not explicitly allowed by our statement of the sequential algorithm. In general, the more freedom of choice there is in the sequential algorithm, the easier it is to parallelize it.

We also cheated a bit when writing the sequential algorithm. In the marking phase, it would be slightly more efficient to add n to S only if n is not grey. However, in the multiple process marking algorithm, process i must add n to S_i before examining n's color, otherwise another process' resetting of S_i to ϕ may not have the desired effect. This is an example of the general observation that to parallelize an algorithm, the sequential ordering of its operations may have to be changed.

The restrictions necessary to allow multiple mutators strike us as being remarkably natural and elegant. We were able to simply postulate that the mutators must be synchronized, and then use that assumption without really knowing anything about how or why they were synchronized. We feel that there must be some underlying general principle involved, but we do not know what it is. We also do not know how formal proof of correctness techniques can be conveniently applied in this case.

The techniques of parallezing the two phases of the collection algorithm, and of pipelining them, appeared to be quite standard. However, the parallel implementation is not completely equivalent to the original sequential algorithm. Proving its correctness requires knowledge of why the sequential algorithm is correct. The color purple and the coloring step 2 were introduced to solve the general pipelining problem of keeping the overlapping computations from interfering with one another.

Our solution can be viewed as an attempt to optimize the algorithm for execution on a multiprocessor computer. We can apply our experience to program optimization in general to conclude that doing a good job of restructuring a program requires understanding why it works. The reasons why the program works are embodied in the proof of its correctness. We expect that in the future, a programmer will construct a correctness proof with every program. Sophisticated optimizing compilers will make use of this proof.

Ultimately, the programmer will have sophisticated automated assistance in verifying the correctness of his program. Until then, he will have to construct difficult multiprocess algorithms by himself. We believe that the method of parallelizing a simpler sequential algorithm will make this task easier.

References

[1] Dijkstra, E.W., et. al.: "On-the-fly Garbage Collection: An Exercise in Cooperation", to appear in Comm. ACM.

[2] Steele, Guy L., Jr.: "Multiprocessing Compactifying Garbage Collection", Comm. ACM 18, 9 (September 1975), 495-508.

HIERARCHICAL PROPERTIES OF CONCURRENCY

by G. S. Tjaden
Sperry Univac Systems Division
Technical Planning Department
Blue Bell, Pa. 19422

Abstract: Instructions and tasks can be equivalently treated as requests for service by computational resources. For any given machine language program a request hierarchy can be constructed which has interesting applications to the problem of the dynamic hardware detection and control of execution of concurrency. Starting with a binary vector-pair model of instructions and knowledge of the destinations and branch instructions, a hierarchy of tasks is constructed which allows a global dynamic analysis of large programs to be made by the hardware during the execution of the program. This approach could lead to detectable program execution speed-ups on the order of 2^N for an N level hierarchy. Better speed-up results should be obtainable for "top-down structured" programs than for "unstructured" programs.

1.0 Introduction

As pointed out by Amdahl (1), there is a strong economic motivation to produce higher performance less costly computers which do not require new or modified software for efficient operation. Such computers can be designed by 1) using a faster-cheaper technology to implement an existing architecture, 2) designing a new architecture which executes an existing instruction repertoire but utilizes more of the concurrency in existing instruction streams to increase the performance, or 3) some combination of these approaches.

Approach 2 can be applied in many ways. This paper will discuss one of these involving extensions to the currently known methods for the hardware detection and control of execution of concurrency in machine language instruction streams during the execution of the streams (dynamically).

An instruction can be thought of as a request for service by computational resources (4). A program or task is then a stream of such requests. On the other hand, a program or task itself is a request for service by computational resources. The task is composed of a stream of "sub-requests" for service, each of which may also be a task composed of further "sub-requests" as shown in Figure 1. Thus, from the viewpoint of requestors and servers, instructions and tasks become indistinguishable. A problem as embodied in a program or task can then be thought of as a "request hierarchy", with only the bottom level of the hierarchy being requests for the actual hardware resources of the computer.

This request hierarchy has certain interesting properties with respect to the dynamic detection and control of execution of concurrency. In particular, this hierarchical representation can allow a global analysis for concurrency detection purposes of a large program dynamically with the hardware. Such an analysis has normally been thought of as being feasible only with software during a "pre-processing" phase. It is also possible, through proper construction of the request hierarchy, to "remove" branch instructions from the instruction (request) stream with the expectation of improving the detectable instruction independencies in the stream.

2.0 Representation of Instructions and Tasks

2.1 Definitions. Computers are thought of as being composed of two types of resources.

Type 1: Storage resources (s-resources), which preserve values over time.

Type 2: Transformational resources (t-resources) which transform values obtained from storage resources (the sources of the t-resource) and place the results into storage resources (the sinks of the t-resource).

Resources are used to perform computations. Computations are specified by instructions.

Definition 1: An instruction I is as follows:

a) A specification of a set of transformational resources, a set of sources for these transformational resources and a set of sinks for these transformational resources.

b) An ordering relation (partial or total) over the set of transformational resources.

Complex computations generally require more than one instruction for their specification. Such complex computations are specified by tasks.

Definition 2: A task T is as follows:

a) A specification of a set of instructions.

b) An ordering relation (partial or total) over this set of instructions.

Let the set of instructions specified by the task be indexed by the positive integers so that I_i is a particular instruction and $1 \leq i \leq N$, where N is the number of instructions in the task. Let the ordering relation $\circledS$ be interpreted such that if $I_i \circledS I_j$, then I_i must appear in the sequence before I_j. For a partial ordering relation it may be the case the $I_i \circledS I_j$ and $I_i \circledS I_k$, but $I_j \not\circledS I_k$ and $I_k \not\circledS I_j$ ($\not\circledS$ means no ordering is defined). In this case more than one initial execution sequence is defined. That is, the sequences I_i, I_j, I_k, and I_i, I_k, I_j are both initial sequences under the above ordering relation. The fact that I_j and I_k are nor ordered with respect to each other and that tasks must be deterministic implies that these instructions may be executed at the same time (concurrently) or in any order and still preserve determinacy.

If the ordering relation is total, then only one initial execution sequence is defined, called here the serial execution sequence. Although concurrent execution cannot occur under a total ordering relation, a partial ordering relation can be derived such that the same values are computed under the partial ordering as under the total. The term potential concurrency will be used to refer to the chances for concurrent execution under an ordering relation.

Execution of a task under an ordering relation involves an interaction between the ordering relation and the instructions specified. That is, the actual sequence in which executions are made may be different from the initial sequence defined by the ordering relation. This difference is because the execution of branch instructions can cause the ordering $I_i \circledS I_j$ to be altered. Reference (7) formally classifies branch instructions into two types, forward and backward by how they alter the orderings of the relation. Branch instructions are informally characterized here by the relative position of the instruction to which the branch instruction transfers control, called the destination of the branch instruction. If a branch instruction I_i has a destination I_d, $d \neq i + 1$, in the serial execution sequence such that $i < d$, I_i is a forward branch instruction. Otherwise it is a backward branch instruction.

Backward branch instructions can cause certain subsequences of the initial serial execution sequence to be executed more than once. Thus, these subsequences may appear more than once in the actual execution sequence.

Definition 3. A cycle is any subsequence of the initial serial execution sequence that appears more than once in the actual execution sequence. Each occurrence of a cycle is called an iteration of the cycle.

Conversion of a totally ordered task to a partially ordered one must be done in such a way that determinacy of the resulting execution sequences with respect to the original serial sequence is preserved. The following definition is the key to converting total ordering relations into partial ordering relations.

Definition 4. Two instructions, I_i and I_j are independent if and only if no sink of I_i is a source of I_j and no sink of I_j is a source of I_i. Otherwise I_i and I_j are dependent.

When I_i and I_j are dependent, a dependency is said to exist between them. From Definition 4, dependencies exist when a sink of one instruction is a source of the other. Dependencies are here classified into two types, data and procedural. Procedural dependencies are caused only by branch instructions, while data dependencies can be caused by both branch and nonbranch instructions. Branch instructions are thought of as calculating values which either deactivate or reactivate certain orderings in the ordering relation.

Definition 5. Suppose that the s-resource denoted by r_x is a sink of I_i and a source of I_j. Then there is a dependency between I_i and I_j. If I_i is is a branch instruction and r_x is the sink used by I_i for the values which effect orderings, then the dependency is a procedural dependency. Otherwise the dependency is a data dependency. Procedural dependencies must be treated differently from data dependencies. This difference in treatment is because data dependencies indicate the necessity of observing a specific order of execution, while procedural dependencies indicate that there is an uncertainty as to whether or not an instruction should be executed. The s-resources into which branch instructions place deactivation-reactivation values are called IC resources.

There is a special type of independency caused by backward branch instructions. Instructions that belong to the same cycle, but are independent in different iterations of the cycle will be called cyclically independent. Techniques for detecting this intercycle dependence are complicated by the fact that, in general, only after the execution of a backward branch is it known if another iteration of a cycle should be executed. Thus, this detection must be done dynamically (that is, while the task is being executed).

2.2 Vector Representation and Properties

Detection of independence of instructions requires knowledge of the source and sink resources of the instructions. Let the storage resources be indexed by the positive integers so that each s-resource has a unique index. The symbol r_i will be used to refer to the s-resource whose index is i. For any instruction I_j two binary vectors $\hat{e}_j$ and $\hat{d}_j$ are defined as follows:

$$\hat{e}_{ji} = \begin{cases} 1 & \text{iff } r_i \text{ is a sink of } I_j \\ 0 & \text{otherwise} \end{cases}$$

$$\hat{d}_{ji} = \begin{cases} 1 & \text{iff } r_i \text{ is a source of } I_j \\ 0 & \text{otherwise} \end{cases}$$

Thus, the set of storage resources are thought of as a resource space and the vectors $\hat{e}_j$ and $\hat{d}_j$ for each instruction I_j are vectors in this space.

For the purposes of this paper, instructions will be considered to be completely characterized by these vectors $\hat{d}$ and $\hat{e}$. This characterization allows the independence (and dependence) of two instructions to be expressed mathematically. The following lemma follows trivially from Definition 4.

Lemma 1: Two instructions I_i and I_j are independent iff $\hat{e}_i \cdot \hat{d}_j = \hat{e}_j \cdot \hat{d}_i = 0$, and are dependent otherwise.

It is assumed that the multiplication indicated is the Boolean scaler product operation.

Reference (2) describes in detail a method by which potential concurrency can be detected and concurrent instruction execution controlled using the source-sink vector pairs for the instructions of a task. This particular method involves the computation of an "ordering matrix" (similar to a precedence matrix) for concurrency detection and control. Several different types of ordering matrices, differing in the amount of potential concurrency detectable, are discussed. It is shown that intercycle independencies can be detected and controlled with a properly constructed ordering matrix.

The particular method used to detect and control concurrency is not really important to this paper. It is important to have shown, however, that a description of a task using source-sink vector pairs and an initial serial ordering is sufficient for concurrency detection and control. Certain of the empirical results discussed in Reference (2) with respect to the measured potential currency obtainable with ordering matrices will be used to infer an expected potential concurrency obtainable with the hierarchical approach of this paper.

3.0 Formation of Levels

3.1 A First Approach. An instruction at level V_1, I_i^1, can be formed from a task at level V_o, T_i^o, in the following way. The set of storage resources which the level V_o instructions have as sources or sinks are said to form a resource space.

The set of $\hat{d}$ and $\hat{e}$ vectors of these level V_o instructions is said to define a Boolean Vector Space, also denoted by the symbol V_o. Thus, associated with a level of instructions, V_i, is a Boolean Vector Space (BVS), V_i. This vector space will not be Euclidean because of the way operations in this space will be defined. Thus, here it is called Boolean.

To form I_i^1 from T_i^o it is sufficient to find two vectors, $\hat{d}_i$ and $\hat{e}_i$, which completely define the sources and sinks of I_i^1 in V_1. The space V_1 is formed from V_o by partitioning of the resources of V_o into sets. Each of these sets is a single resource of V_1.

Although any arbitrary partitioning of the resources of V_o into disjoint sets would produce a valid space V_1, a particular partitioning scheme which facilitates construction by a preprocessor (before any executions take place) will be described.

Suppose a large task, T^o, of size N is partitioned into subtasks of size n. Let these subtasks be denoted by T_1^o, T_2^o, ..., $T^o_{\{\frac{N}{n}\}}$. Level V_o instructions $I_1 \ldots I_n$ are in T_1^o, $I_{n+1} \ldots I_{2n}$ are in T_2 etc. Define the vectors

$$\hat{e}_i' = \bigvee_{j=n.(i-1)+1}^{n.i} \hat{e}_j \text{ and } \hat{d}_i' = \bigvee_{j=n.(i-1)+1}^{n.i} \hat{d}_j$$

That is, $\hat{e}'_i$ and $\hat{d}'_i$ are the vectors formed by taking the element by element union of the sink and source vectors of the instructions of T^o_i. The vectors $\hat{e}'_i$ and $\hat{d}'_i$ are in the space V_o.

The sets of vectors $\{\hat{e}'_i\}$ and $\{\hat{d}'_i\}$ contain all of the resource information necessary to correctly order the relative execution of the subtasks. In fact, one could use these vectors to calculate an ordering matrix for these subtasks. Each subtask would be treated as an instruction, and V_1 would be a subspace of V_o. Potential concurrency would be lost, as the following example demonstrates.

Suppose I_x in T^o_i has r_k as a sink, and no serially previous instruction in T (remember T^o_i is a subtask of T) has r_k as a source or a sink. Also suppose that I_{x+a} in T^o_i has r_j as a source, and that T^o_{i-b} contains an instruction having r_j as a sink. Neglecting branch instructions I_x would be executably independent as soon as T is activated, but T^o_i would not be executably independent until T^o_{i-b} is executed because of the dependency caused by r_j.

The space V_o is a very impractical one with which to work. For a large task, T, the dimension of V_o would be very large since each storage resource specified occupies one component position in the vectors of V_o. Notice, however, that each subtask, T_i, specifies only a subset of the s-resources specified by T. By assigning this subset of resources to a single component position in the vectors of another space, V_1, the dimension of the vectors in V_1 can be reduced. If the dimension of V_1 is still too large, the space, V_1, can be partitioned into another space, V_2, of even smaller dimension. This process can continue until a space has been constructed whose dimension satisfies any arbitrary constraint.

A procedure for constructing these spaces is now given. Because of the relationship between tasks and instructions, this single procedure is used recursively to construct all of the higher level spaces in a single pass over the level V_o task. An important parameter in this procedure is the decision rule for determining the partitioning of a task into subtasks. For ease of understanding, it will be assumed for now that the partitioning is only according to the size of the subtask. That is, starting at the first instruction of a task I_1, it is partitioned into subtasks of equal size, n. It is assumed that, when the procedure is examining instruction I_i of the task only the subspace of V_o defined by the instructions previous to I_i in the initial execution sequence is known to the procedure. Thus, the space, V_o, is completely defined only <u>after</u> the procedure has terminated. The following procedure is illustrated in Figure 2.

(1) Begin by constructing $\hat{d}_1$ and $\hat{e}_1$ for I_1, in the subspace of V_o defined by I_1. That is, assign each source and sink of I_1 to a component position in V_o, and set the components of $\hat{e}_1$ and $\hat{d}_1$ to one or zero as required. The assignments of resources to component positions are stored in a special table called the Resource Table (RT). If this procedure were incorporated into a compiler, the RT could be part of the symbol table of the compiler.

(2) Then construct the vectors $\hat{d}_2$ and $\hat{e}_2$ for I_2. For each resource, r_x, specified by I_2, check the resource table to determine to which component position r_x has been assigned. If r_x has not been assigned a position (because it was not requested by I_1) then a position is assigned to it, this assignment is noted in the RT, and the components of $\hat{d}_2$ and $\hat{e}_2$ are set accordingly.

It is necessary at this point to construct a new pair of vectors.

Define: $\underline{\hat{d}}^o_1 = \hat{d}_1 \vee \hat{d}_2$ and $\underline{\hat{e}}^o_1 = \hat{e}_1 \vee \hat{e}_2$. It is these vectors which will be used to construct the space V_1. the superscript of $\underline{\hat{d}}^o_1$ denotes the fact that this vector is formed at level V_o, and the subscript indicates that it was formed from instructions in subtask T^o_1 (first n instructions in T). See Figure 2 part b.

(3) Construct, successively, the vectors $\hat{d}_i$ and $\hat{e}_i$ for each I_i such that $3 \leq i \leq n$. After each pair of vectors is constructed, perform the operations

$\hat{\underline{d}}_{1}^{o} := \hat{\underline{d}}_{1}^{o} \vee \hat{d}_i$ and $\hat{\underline{e}}_{1}^{o} := \hat{\underline{e}}_{1}^{o} \vee \hat{e}_i$. Then proceed to I_{i+1}. See Figure 2 part c.

(4) After the source and sink vectors for I_n have been constructed, the construction of the Boolean Vector Space, V_1 may be started. This is done by partitioning the set of resources specified in T_1^o into disjoint subsets, and assigning a single resource in V_1 to each of these subsets. There is no a priori reason to suspect that any rule for forming these subsets is better than any other. The simplest rule would be to not form any subsets at all. That is, assign all of the resources of T_1^o to a single component in the vectors of V_1. Another rule might be to form two subsets, one of which has all of the sources which T_1^o specifies (for which the elements of $\hat{\underline{d}}_{1}^{o}$ are set to one), and the other having the sinks (if any). One would expect, intuitively, that partitioning into many subsets will result in a smaller decrease in potential concurrency than partitioning into only a few (or none) subsets. The dimension of V_1 will grow larger as more subsets are formed, however.

To simplify the explanation, we choose to assign the set of resources specified by T_1^o to a single resource in V_1. Before describing the modifications to the RT necessary to implement this rule, the special way in which the IC storage resource, r_{IC}, is handled must be described. An IC resource is defined to exist at every level. Thus, the IC resource is not included in the set of V_o resources assigned to a single V_1 resource. The IC resource is assigned to component position one of each resource space.

The modifications necessary to the RT are illustrated in Figure 3 part a. Another row, C_1, must be provided in the table for the resource assignments of space V_1. Under our partitioning rule, element IC of this row is given the value one, and all other elements requested by T_1^o are given the value two. In general, for each resource space formed, V_k, a new row, C_k, must be added onto the RT.

It only remains in this step to form the vectors $\hat{d}_1^1$ and $\hat{e}_1^1$ for the newly created level V_1 instruction I_1^1 corresponding to T_1^o. This is done by using the vectors $\hat{\underline{d}}_{1}^{o}$ and $\hat{\underline{e}}_{1}^{o}$ in conjunction with the RT.

Let RT_{mn} be the value in the nth column of row C_m of the RT for $m \geq 0$ and $n \geq 1$. Then $\forall k$ such that $\hat{\underline{d}}_{1k}^{o} = 1$, find the p such that $RT_{0,p} = k$. Then find $RT_{1p} = x$ and set $\hat{d}_{1x}^{1} = 1$. Set all other elements of $\hat{d}_1^1$ to zero. Then construct $\hat{e}_1^1$ from $\hat{\underline{e}}_{1}^{o}$ in the same way. See Figure 3 part a for an example. These two vectors completely represent I_1^1. They are stored for later use in calculating level V_1 ordering matrices.

At this point, delete the vectors $\hat{\underline{d}}_{1}^{o}$ and $\hat{\underline{e}}_{1}^{o}$ as they are no longer needed. Also, delete the values of the elements on row C_o. These values will be reassigned during the scanning of T_2^o.

(5) At this point in the procedure the vectors of T_2^o in space V_1 are constructed using steps (1), (2) and (3). The level V_o resources of the instructions of T_2^o can be assigned to the same component position in V_o as were used for the instructions of T_1^o, because all orderings between instructions in T_1^o and those in T_2^o will take place via level V_1 instructions I_1^1 and I_2^1. Thus, instructions in different subtasks at the same level may have different resources assigned to the same component position of their respective resource spaces. Figure 3 part b illustrates this, using the instructions $I_5 \ldots I_8$ as T_2^o. Notice that resource C has not been assigned a component position in the space for T_2^o since it is not specified in this subtask.

After the vectors for I_{2n}^o have been constructed a level V_1 instruction, namely I_2^1 is constructed using the method given in step (4). In space V_1 the resources requested in T_1^o have already been assigned a component position. This position is found in row C_1 of the RT. All resources specified in T_2^o but not specified in T_1^o are assigned to a previously unassigned component position in V_1. In the example of Figure 3, the variables R_1, A, B, and C were previously assigned to position 2 in V_1.

To specify I_2^1 its source and sink vectors, $\hat{d}_2^1$ and $\hat{e}_2^1$ must be constructed using step (4) with $\hat{d}_{-2}^o$ and $\hat{e}_{-2}^o$.

The vectors $\hat{d}_2^1$ and $\hat{e}_2^1$ are stored as the second instruction of the level V_1 task. The vectors $\hat{d}_{-2}^1 = \hat{d}_1^1 \vee \hat{d}_2^1$ and $\hat{e}_{-2}^1 = \hat{e}_1^1 \vee \hat{e}_2^1$ are formed for later use in constructing I_1^2 from T_1^1, just as I_1^1 was constructed from T_1^o.

(6) Thus, the construction of instructions at each level is done with an identical procedure. The procedure is applied to each subtask T_i^o for $1 \leq i \leq n$. The space V_1 is formed simultaneously with the formation of V_o. After subtask T_n^o has been scanned (after encountering instruction I_n^o2) there will have been constructed enough level V_1 instructions, I_1^1, I_2^1, ..., I_n^1 to begin forming the space V_2. This space is formed from V_1 in exactly the same way in which V_1 was formed from V_o. Another row in the RT, called C_2, will be needed.

This top-down scanning of T continues and, by recursively applying steps 1-6, successively higher levels are formed. After I_{n^3} is encountered one can begin forming V_3, after I_{n^4} one can begin forming V_4, etc. After all of the instructions of T have been analyzed, there will be a hierarchy of levels. The top level, V_k, will have no more than n instructions, and can thus be represented with an ordering matrix having no more than n rows and columns. Note that the levels are all formed simultaneously, in a single scan of T.

One can see that for all levels, V_i such that $i > 0$, the dimensions of each of the spaces constructed is bounded at n+1. In each space n instructions were constructed, and each instruction adds at most one resource (component position) to the space. The IC storage resource is defined separately in each space, giving at most n+1 components in the vectors of each space. For cases of practical interest, the dimension of V_o is also of bounded size. For example, if the instructions of V_o are machine language instructions, then the resources specified by these instructions consist of a fixed set of machine registers and other physical devices (e.g. I/O channels), plus a fixed number (usually one) of memory cell specifications.

3.2 Some Better Partitioning Rules

If an instruction, I_i^o, in a subtask, T_j^o, is a branch instruction (has r_{IC} as a sink), then the instruction constructed for T_j^o in space V_1 will also be treated as a branch instruction in V_1. Suppose that the average number of instructions between branch instructions in T is u. Also suppose that n is chosen such that $n >> u$. Then the probability that a particular subtask T_j^o, contains at least one branch instruction is very large. But this would mean that almost every level V_1 instruction is a branch instruction, as would also be the case for all levels above V_1. One would expect this proportion of branch instructions to seriously degrade the potential concurrency at these higher levels.

It is apparent that rules for partitioning a task into subtasks such that the probability of creating a higher level branch instruction is minimized (or at least reduced) would be helpful. We will outline here a set of partitioning rules which will help achieve the above goal.

There are two types of instructions whose occurrence during the scanning of T^o will signal a "good" partition point in the sense that there may be a reduction in the number of higher level branch instructions. They are (1) a branch instruction, and (2) a destination instruction. Destination instructions can be detected during assembly by the occurrence of a label.

The following assumptions are made here:

(1) For each instruction, I_i, at whatever level, it is known whether or not I_i is a destination.

(2) If I_i is a branch instruction, then it is known whether I_i is a forward or a backward branch instruction.

(3) If I_i is a destination instruction, then it is known whether I_i is the destination of a forward branch instruction (called a <u>forward destination</u>) or of a backward branch instruction (a backward destination).
Note: It is possible for I_i to be both a forward and a backward destination.

We wish to take advantage of the following type of situation. Suppose I_i^o is a forward branch to I_j^o. If I_i^o and I_j^o are in the same subtask, T_k^o, then I_k^1 need not be designated a branch instruction (at least due to the presence of I_i^o) since at level V_1 the destination of I_k^1 is itself. One can see that good partition points would be:
(1) immediately <u>before</u> a forward branch instruction.
(2) immediately <u>after</u> a backward branch instruction.
(3) immediately <u>after</u> a forward destination.
(4) immediately <u>before</u> a backward destination.

Partitioning at these points will increase the probability of placing a branch instruction and its destination in the same subtask. These partition points will also be good ones to use at higher levels since there will always be some level at which a branch instruction and its destination can be found in the same subtask. However, it may be possible that several branch instructions are in the same subtask, T_i^k, at a particular level. The corresponding level V_{k+1} instruction, I_i^{k+1}, is thus a branch instruction which could have both forward and backward destinations. However, at some higher level, V_{k+p}, this instruction will be in the same subtask as its destinations, and thus will cease to cause branch instructions at levels higher than V_{k+p}.

4.0 <u>Executing Programs From the Hierarchy</u>

We now illustrate one possibility for a computer organization which takes advantage of the existence of this hierarchy. It is assumed that a program is stored in memory as a serially ordered sequence of instructions, and that this sequence has been partitioned into a hierarchy of levels according to the previous section. The higher level instructions are grouped together into their respective levels and are stored as serially ordered sequences in a way which facilitates accessing a subtask of a particular level, T_i^j, knowing only the level V_{j+1} instruction, I_i^{j+1}, to which it corresponds.

The execution of a task, T, will take place as follows. Suppose that V_k is the highest level formed for T. Then the first step is to form an ordering matrix for T^k, called M^k, and to initialize the control variables for M^k using the control variable transition rules of Reference (7). Suppose that two instructions, I_i^k and I_j^k, in T^k are found executably independent. Since these instructions are "higher level" instructions they do not directly request specific physical machine resources. "Execution" of these instructions is done by calculating ordering matrices, for the level V_{k-1} subtasks, T_i^{k-1} and T_j^{k-1}, to which these instructions correspond. Then these subtasks are executed from their ordering matrices. When all of the instructions of T_i^{k-1}, for example, are found inactive (task T_i^{k-1} has terminated) then M^k is notified that I_i^k has completed execution and the transition rules are applied to M^k so that other instructions may be found executably independent.

Suppose that $k \geq 2$. Then the instructions of T_i^{k-1} and T_j^{k-1} must be executed by forming ordering matrices for the level V_{k-2} subtasks to which they correspond, and executing from these ordering matrices. It is only level V_o instructions which are executed directly on the physical machine resources. Notice that if at some level, V_p, for $p > 0$, more than one instruction is found executably independent at the same time, then there will be at least two level V_o subtasks executing concurrently. Each of these subtasks may, of course, have several instructions executably independent at any one time.

No direct measurements have been made to determine the amount of detectable potential concurrency which exists at levels above level zero. There is encouraging indirect evidence, however from which arguments can be made that this hierarchial approach has good potential.

The first such argument is based on the fact that the empirical results of References (5,6) were based on measurements of machine language programs, while those of Reference (2) were based on measurements of high level language programs, (ALGOL and FORTRAN). Since each statement of a high level language task is compiled into an ordered set of machine language instructions, it seems reasonable to consider these machine language "subtasks" as residing at level V_o, while the high level language statements are the equivalent level V_1 instructions. The empirical results show that the average rate of independence (average number of instructions concurrently executable) is about the same (1.8) for both levels. Thus, executing FORTRAN and ALGOL programs under a two-level hierarchy as described here should lead to an effective rate of independence of $1.8 \times 1.8 = 3.24$ (neglecting overheads). Table 1 summarizes the results of Reference (5), labeled Tjaden and Flynn, and Reference (6), labeled Riseman and Foster, as well as Reference (2).

The second argument is based on the strong correlation between the density of branch instructions and the rate of independence shown in Table 1. Assuming that these two factors are correlated as the Table indicates (i.e., branch instruction density inversely proportional to rate of independence), the rate of independence at the upper levels of the hierarchy should be the same as that at lower levels if the branch instruction density can be maintained equal to this density at the lower levels. The partitioning rules and the definition of higher level branch instructions were defined with this maintenance of density in mind.

It is expected that the choice of maximum partition size, choice of partition points, and the inherent control structure of the program will strongly affect the rate of independence at the upper levels. If the partition size is too small, relatively few branch instructions will have destinations located in the subtask with the branch instruction. If the control structure is very random, the chances of finding a partition point such that all of the branch instructions in a subtask also have destinations in that subtask will be relatively low. Top-down structured programs should be well suited for execution in the hierarchical environment described here. Not only are such programs hierarchical in structure themselves, but their reliance on procedure calls and limited control structure forms should reduce the density of branch instructions at all levels. Procedure calls, in particular, should not be treated as branches and returns, but as higher level instructions with the procedure itself treated as a subtask at a lower level.

5.0 Conclusion

If an average rate of independence of two can be achieved at each level of the hierarchy, then the effective rate of independence should be on the order of 2^N, where N is the number of levels. Very large problems should result in a hierarchy having a great number of levels, and a resulting large effective rate of independence. There is, of course, a maximum number of levels beyond which no increase in potential concurrency will be realized. Riseman and Foster (6) have measured a maximum average speed-up (rate of independence) due to concurrency of about fifty. Thus, the maximum number of useful levels will be that number, N, such that $2^N \leq 50$, or $N = 5$. If, for example, a partition size of 32 is chosen, the largest program which can be partitioned under the 2^N speed-up can have no more than $2^{25} \simeq 3 \times 10^7$ instructions. If two memory words are required to store the source-sink vector pair for each subtask (partition) of size thirty-two, the memory overhead as a percentage of program size will be about 7%.

It is very reasonable that the rate of independence should increase rapidly and nonlinearly with program size using the hierarchical approach. One would expect large programs to possess a correspondingly higher potential for concurrency than smaller programs because of global independencies. This hierarchical approach can result in detection of these global independencies. It is not clear that present dynamic concurrency detection algorithms are powerful enough to detect such global independencies. One would expect that algorithms which use semantic information contained in DO-LOOP definitions, for example, may be required to effectively utilize global concurrency. A good deal of research remains to be performed before the viability of these ideas can be determined.

REFERENCES

(1) G. M. Ahmdahl, Keynote Address of The Third Annual Symposium on Computer Architecture, (January 19, 1976).

(2) G. S. Tjaden and M. J. Flynn, "Representation of Concurrency with Ordering Matrices," IEEE-TC, (August, 1973), pp. 752-761.

(3) R. M. Keller, "Look-Ahead Processors," ACM Computing Surveys, (December, 1975), pp. 177-196.

(4) M. J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE-TC, (September, 1972), pp. 948-960.

(5) G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," IEEE-TC, (October, 1970), pp. 889-895.

(6) E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," IEEE-TC, (December, 1972), pp. 1405-1410.

(7) G. S. Tjaden, "Representation and Detection of Concurrency Using Ordering Matrices," Ph.D. Dissertation, The Johns Hopkins University, Baltimore, Md. (1972).

FIGURE 1

The Relationship of Tasks and Instructions

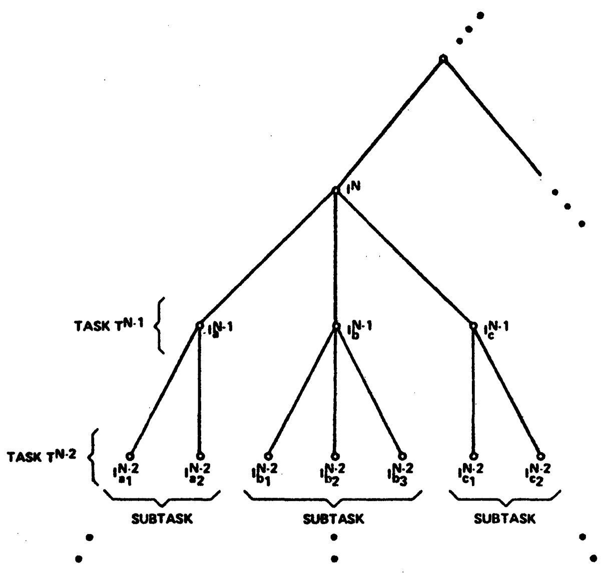


FIGURE 2

Forming Resource Space V_O

I_1: $R_1 = A$
I_2: $R_1 = R_1 + B$
I_3: IF $R_1 > 0$ GO TO X
I_4: $C = R_1$

PART A

SUBTASK T_1^0 FOR $n = 4$

RESOURCE TABLE

	RESOURCE	IC	R_1	A	B	
row C_0	COMPONENT	1	2	3	4	

	IC	R_1	A	B
$\hat{d}_1^0 =$	0	0	1	
$\hat{d}_2^0 =$	0	1	0	1
$\underline{\hat{d}}_1^0 =$	0	1	1	1

	IC	R_1	A	B
$\hat{e}_1^0 =$	0	1	0	
$\hat{e}_2^0 =$	0	1	0	0
$\underline{\hat{e}}_1^0 =$	0	1	0	0

PART B
AFTER STEP (2)

FIGURE 2

Forming Resource Space V_O (Cont'd)

RT

RES.	IC	R_1	A	B	C	
C_0	1	2	3	4	5	

	IC	R_1	A	B	C
$\underline{\hat{d}}_1^0 =$	0	1	1	1	
$\hat{d}_3^0 =$	0	1	0	0	0
$\underline{\hat{d}}_1^0 =$	0	1	1	1	0
$\hat{d}_4^0 =$	0	1	0	0	0
$\underline{\hat{d}}_1^0 =$	0	1	1	1	0

	IC	R_1	A	B	C
$\underline{\hat{e}}_1^0 =$	0	1	0	0	
$\hat{e}_3^0 =$	1	0	0	0	0
$\underline{\hat{e}}_1^0 =$	1	1	0	0	0
$\hat{e}_4^0 =$	0	0	0	0	1
$\underline{\hat{e}}_1^0 =$	1	1	0	0	1

PART C
AFTER STEP (3)

FIGURE 3

Forming Resource Space V_1

RT

RES	IC	R_1	A	B	C	
C_0	1					
C_1	1	2	2	2	2	

$$I_1^1 \begin{cases} \hat{d}_1^1 = 0 \quad 1 \quad . \quad . \quad . \\ \hat{e}_1^1 = 1 \quad 1 \quad . \quad . \quad . \end{cases}$$

PART A
MODIFICATION OF THE RT TO FORM
SPACE V_1 FROM STEP (4)

FIGURE 3

Forming Resource Space V_1 (Cont'd)

$$T_2^0 \begin{cases} I_5\text{:} \quad R_1 = D \\ I_6\text{:} \quad R_1 = R_1 * A \\ I_7\text{:} \quad R_2 = R_1 + B \\ I_8\text{:} \quad D = R_2 \end{cases}$$

RT

RES.	IC	R_1	A	B	C	D	R_2	
C_0	1	2	4	6		3	5	
C_1	1	2	2	2	2	3	3	

	IC	R_1	D	A	R_2	B
$\hat{d}_5^0 =$	0	0	1			
$\hat{d}_6^0 =$	0	1	0	1		
$\hat{d}_7^0 =$	0	1	0	0	0	1
$\hat{d}_8^0 =$	0	0	0	0	1	0
$\hat{\underline{d}}_2^0 =$	0	1	1	1	1	1
$\hat{d}_2^1 =$	0	1	1	.	.	.

	IC	R_1	D	A	R_2	B
$\hat{e}_5^0 =$	0	1	0			
$\hat{e}_6^0 =$	0	1	0	0		
$\hat{e}_7^0 =$	0	0	0	0	1	0
$\hat{e}_8^0 =$	0	0	1	0	0	0
$\hat{\underline{e}}_2^0 =$	0	1	1	0	1	0
$\hat{e}_2^1 =$	0	1	1	.	.	.

PART B
THE RESOURCE SPACES FOR T_2^0 AND I_2^1

TABLE 1

TASK	410	417	428	TOTALS	AVERAGE	
NO. INSTS.	62	48	58	168		
NO. INSTS. EXECUTED	173	102	233	608		
DENSITY OF BRANCH INST.	0.371	0.354	0.241			
TEST 1 $(E \cdot Y) v (E \cdot Y)^t$	1.21	1.22	1.64		1.36	RATE OF INDEPENDENCE
TEST 2 $(E' \cdot Y') v (E' \cdot Y')^t$	1.4	1.59	1.83		1.61	
TEST 3 $(E' \cdot Y') \Diamond (E' \cdot Y')^t$	1.5	1.67	2.2		1.79	
TEST 4 $(E' \cdot Y') \Diamond (E' \cdot Y')^t - (R')^2$	1.5	1.67	2.33		1.83	
TEST 5 $M^d \, VM^p \, VM^{pp}$	1.53	1.96	2.45		1.98	

THE FLIP NETWORK IN STARAN [a]

Kenneth E. Batcher
Digital Technology Department
Goodyear Aerospace Corp.
Akron, Ohio 44315

Abstract - The flip network in each array module of STARAN scrambles and unscrambles multi-dimensional access (MDA) memory data. The flip network can permute data on transfers from memory to PE's, from PE's to memory, and from PE's to PE's. Among the allowable permutations are barrel shifts, barrel shifts on substrings, and FFT-butterflies. The network can be used for such data manipulations as shifting, mirroring (flipping end-for-end), irregular spreading, or compressing and replicating. These manipulations are useful for sorting, fast Fourier transforms, image warping, and solving partial differential equations on multi-mesh regions.

Introduction

An earlier paper (Ref. 1) describes the multi-dimensional access (MDA) memories in STARAN. Memory data can be accessed (fetched or stored) by words, by bit-slices, by byte-slices, etc. MDA memories are built with ordinary RAM chips, and data is scrambled a certain way when stored in memory so that it can be accessed in various ways.

A scramble/unscramble network is required to scramble the data when it is stored into memory and to unscramble the data when it is read from memory. The flip network (Figure 1) does the scrambling and unscrambling and can also perform a number of other useful permutations. Bauer (Ref. 2) has shown how a number of data manipulating functions can be performed using the flip network with appropriate PE masking.

Here, we show the construction of the flip network and then a method of irregularly spreading and compressing data that is faster than the method shown in Ref. 2.

Flip Network Construction

Notation

A 2^n -item flip network has 2^n input-data-lines labeled with n-bit binary vectors ranging from (00...00) to (11...11). It has 2^n output-data-lines also labeled with n-bit binary vectors. The network has two control inputs:

1. An n-bit flip control that specifies one of 2^n flip-permutations.

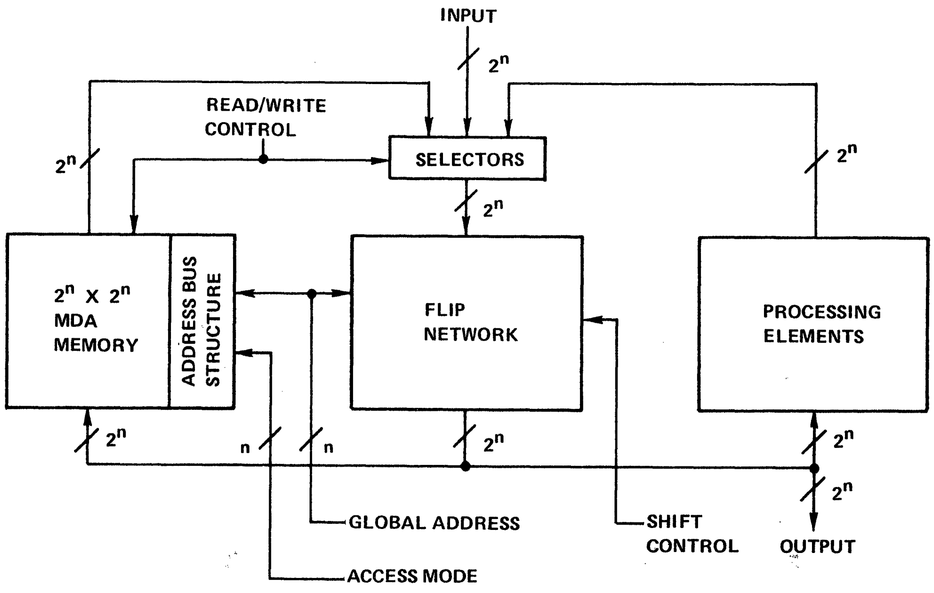


Figure 1. STARAN Array Module (n = 8)

[a]Trademark, Goodyear Aerospace Corporation, Akron, Ohio.

2. A shift-control that specifies one of $(n^2 + n + 2)/2$ shift-permutations.

The flip network permutes the input data first according to the specified flip-permutation, then according to the specified shift-permutation, and presents the permuted data on its output-data-lines.

To scramble and unscramble MDA memory data, the data is fed through the flip network while the flip-control is driven by the MDA memory global address to cause the desired flip-permutation.

Flip-Permutations

If $F = (f_{n-1}\ f_{n-2} \ldots f_1 f_0)$ is the n-bit binary vector fed to the flip-control, the flip-network moves the data on input-data-line $I = (i_{n-1}\ i_{n-2} \ldots i_1 i_0)$ to output-data-line $I \oplus F = (i_{n-1} \oplus f_{n-1}, i_{n-2} \oplus f_{n-2}, \ldots, i_1 \oplus f_1, i_0 \oplus f_0)$, where $\oplus$ means the exclusive-OR logic function.

Figure 2 shows the flip-permutations for an 8-item flip network. When F = (00...00), there is no permutation (the identity permutation); when F = (11...11), there is a complete reversal of data end-for-end (the mirror permutation). Each flip-permutation is its own inverse, and any two permutations commute with each other. If $F = F_1 \oplus F_2$, then flip-permuation F can be performed by doing permutation F_1 followed by F_2.

If the control input F has a single 1 and n-1 0's, then flip permutation F is called an atom (for the 8-item flip network, the atoms are (001), (010), and (100)). The set of n atoms forms a basis for all flip-permutations (any flip-permutation can be formed from atoms). This suggests one way of constructing flip networks. A 2^n-item flip network can be formed from n levels of logic. Each level is controlled by one of the flip-control bits and performs one of the atom permutations whenever the control bit is 1.

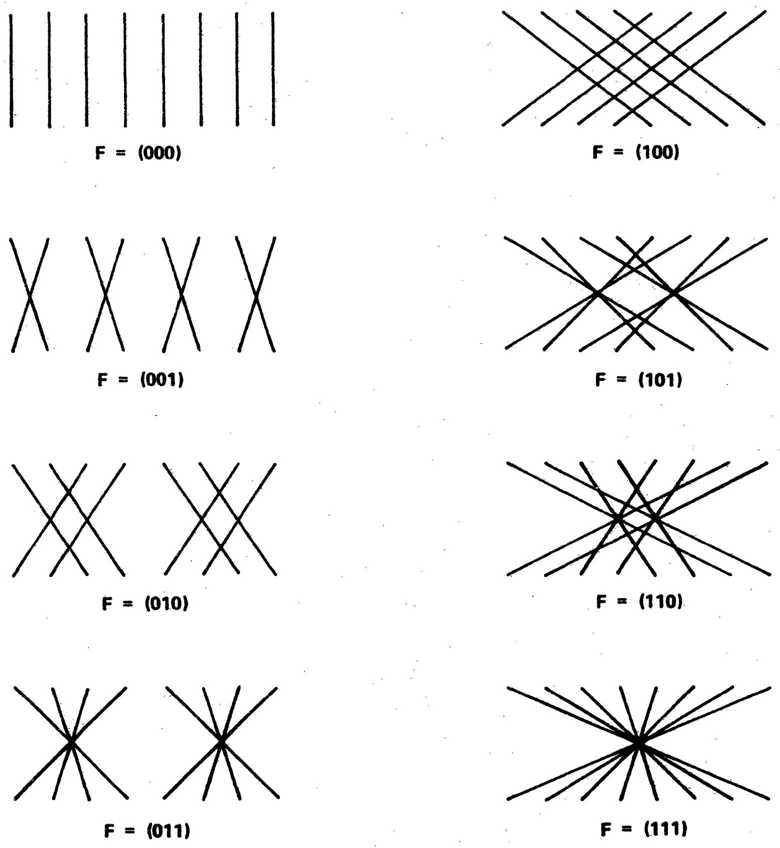


Figure 2. Flip Permutations for 8-Item Flip Network

Figure 3 shows an 8-item flip network constructed this way. The first level of logic performs flip-permutation (001) if the least-significant flip-control bit is 1 and identity permutation if the control bit is 0. Similarly, the second level does flip-permutation (010) when the middle control bit is 1 and the last level does flip-permutation (100) when the most significant control bit is 1. With this construction method, a 2^n-item flip network requires n levels of logic, with each level comprising 2^n two-way data selectors.

Figure 4 is an 8-item flip network redrawn to illustrate that the levels of data selectors are alike when the data is shuffled between levels. This means that a flip network can be built from a number of identical modules. It also means that the data can be recirculated n times through one level of data selectors if it is shuffed at each pass. Thus, one can use a shuffle-exchange network (Ref. 3) as a flip network.

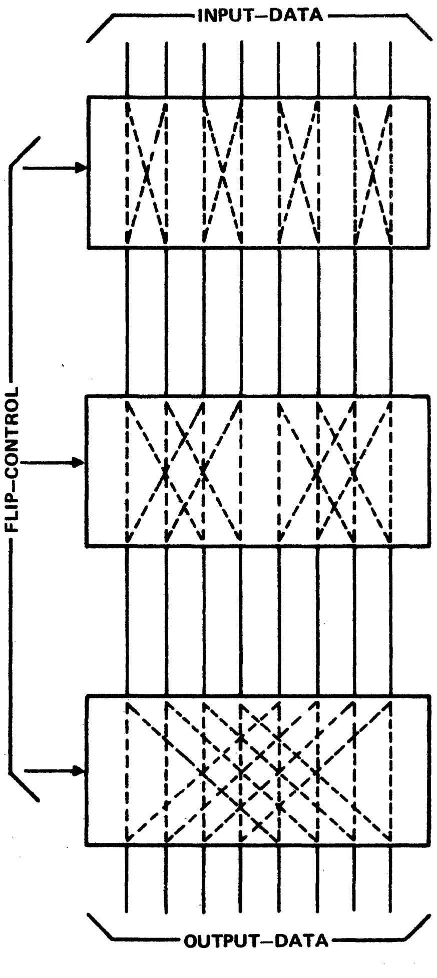


Figure 3. An 8-Item Flip Network

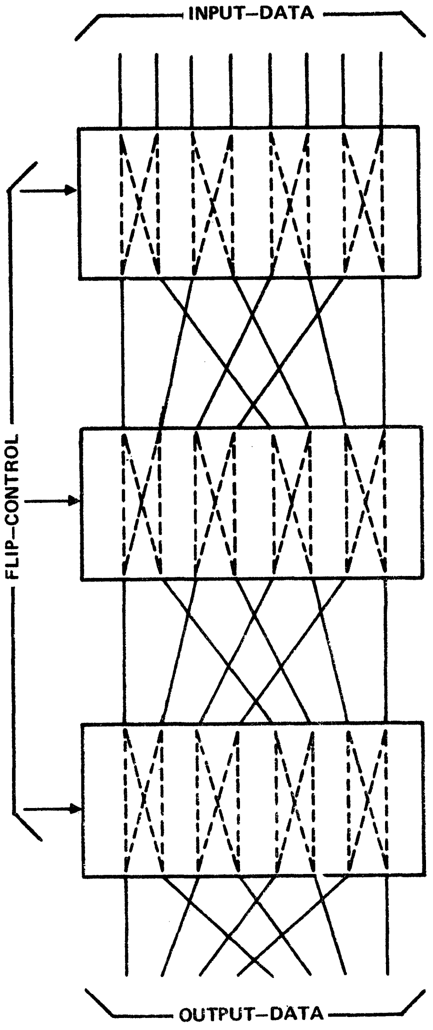


Figure 4. An 8-Item Flip Network Redrawn

One level of four-way data selectors can take the place of two levels of two-way data selectors. If n is even, a 2^n item flip network can be built from n/2 levels of four-way selection. The 256-item flip networks in the current STARAN each have four levels of four-way data selectors.

Shift-Permutations

The shift-control input to a 2^n-item flip network allows one of $(n^2 + n + 2)/2$ shift-permutations to be applied after any flip-permutation. One of the shift-permutations is the identity permutation (no shifting); the other $(n^2 + n)/2$ permutations are shifts of 2^m places modulo 2^p where m and p are integers so that $0 \leq m < p \leq n$. A shift of 2^m modulo 2^p divides the 2^n data items into groups of 2^p items each and shifts the items within each group right end-around 2^m places. Figure 5 illustrates the seven shift permutations in an 8-item flip network.

When $m = p - 1$, the shift-permutation of 2^m modulo 2^p is the same as a flip-permutation (compare the 1 mod 2, 2 mod 4, and 4 mod 8 shift-permutations of Figure 5 with the (001), (010), and (100) flip-permutations, respectively, of Figure 2). Other shift-permutations are performed in the flip network by selectively controlling the data selectors on certain levels. Figure 6 shows how a 1 mod 8 shift-permutation is performed in the 8-item flip network of Figure 3.

The selective control of data selectors on a level required for the shift-permutations is accomplished by expanding the number of control signals for the level; each control signal controls a fixed subset of the selectors on the level. With levels of two-way selectors, the first level has one control signal, the second level uses two

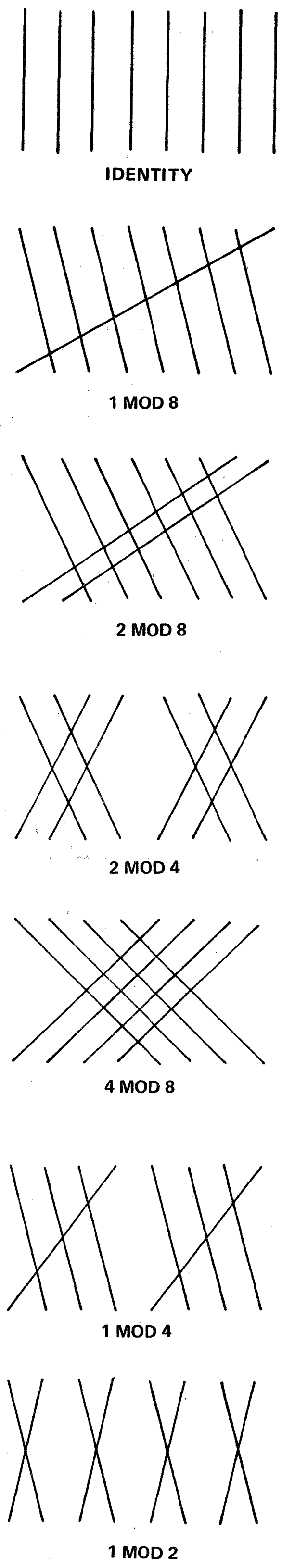


Figure 5. Shift Permutations in an 8-Item Flip Network

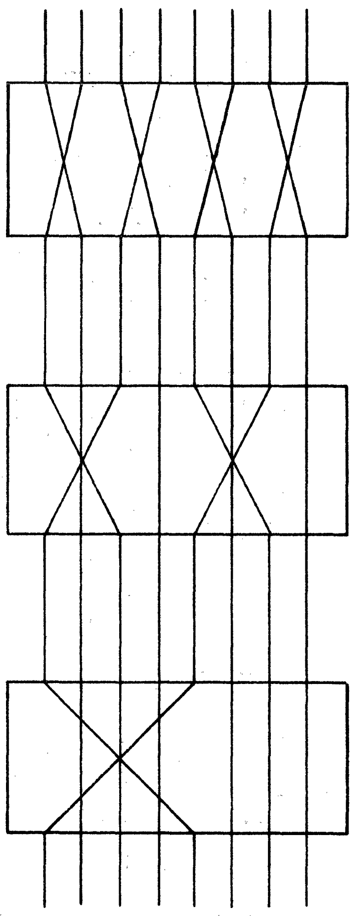


Figure 6. The 1 Mod 8 Shift Permutation in an 8-Item Flip Network

control signals, the third level uses three control signals, etc. A 2^n-item flip network requires n (n + 1)/2 control signals. Figure 7 shows how six control signals control the data selectors of an 8-item flip network so that both flip and shift permutations can be performed. The control table for this network follows (when the control signal is 1, the selectors swap data):

	Control Signal					
Permutation	0A	1A	1B	2A	2B	2C
1 mod 8	1	1	0	1	0	0
2 mod 8	0	1	1	1	1	0
4 mod 8	0	0	0	1	1	1
1 mod 4	1	1	0	0	0	0
2 mod 4	0	1	1	0	0	0
1 mod 2	1	0	0	0	0	0
Identity	0	0	0	0	0	0

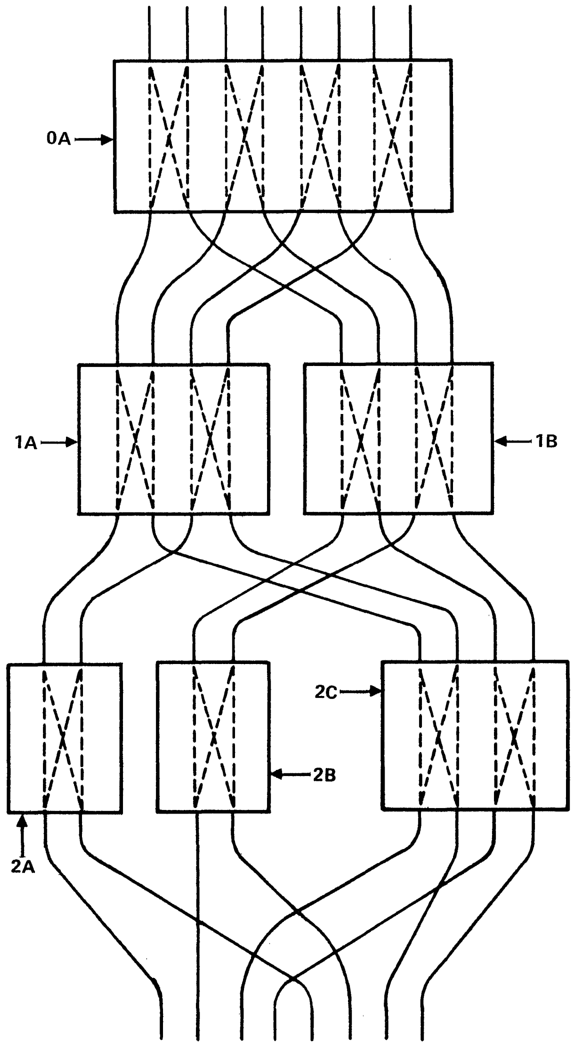


Figure 7. An 8-Item Network for Flip and Shift Permutations

For flip-permutations, 0A is driven by the least-significant flip-control bit; 1A and 1B are driven by the middle flip-control bit; and 2A, 2B, and 2C are driven by the most-significant flip-control bit.

To allow a combination of a flip-permutation with a shift-permutation in one pass through the network, each control signal is driven from an exclusive-OR gate. The shift-permutation control is fed to one exclusive-OR input and the flip-permutation control is fed to the other input. The resultant permutation is the same as the flip-permutation followed by the shift-permutation.

To shift data in a negative direction, one can mirror the data with a flip-permutation (11...11), shift the mirrored data in a positive direction, and then remirror the data with another (11...11) flip. The mirroring and remirroring can be combined with the shifts. For example, a shift of -31 places can be performed in two passes: A mirror with a shift of 32 followed by a mirror with a shift of 1.

Data Manipulations

General

The flip network can permute data on memory-to-PE transfers, PE-to-PE transfers, and PE-to-memory transfers. The permutations are useful in many applications to manipulate or route data between PE's. Bauer (Ref. 2) illustrates a number of these manipulations. Some manipulations require only one pass through the network; several require $\log_2$ N passes for N items. One class of functions (irregular compress and expand) required about N passes for N items. Here, we show how these irregular functions can be accomplished in about $\log_2$ passes.

Irregular Spreading

Spreading (expanding, replicating) takes the output of a contiguous set of PE's and spreads it across a larger set of PE's, replicating some items but preserving their relative order. As an example, if we let

abcde (1)

represent the outputs of the first five PE's in order, then irregular spreading can create the following pattern of 19 items:

aaaa b ccccc ddddddd e (2)

in the first 19 PE's.

Spreading arises in a number of problems. To magnify a digitized image, new picture elements (pixels) are created on a finer grid; the old pixels must be spread and then interpolated to create the new image. This spreading is irregular if the image is being warped. Another example is solving partial differential equations on multi-mesh regions; data computed on a coarse mesh must be spread and interpolated when moved across a boundary to a finer mesh.

In STARAN, spreading is accomplished with shift-permutations in the flip-network combined with appropriate PE masks. It will be illustrated with the example of spreading pattern (1) to obtain pattern (2). Figure 8 shows the state of the first 19 PE's at different steps of the process.

Initially, the five data items (a, b, c, d, and e) are stored in the first five PE's (0, 1, 2, 3, and 4, respectively). Each PE is to receive one of these items. The second column of Figure 8 shows the initial location of the item (e.g., PE's 5 through 9 are to receive item c, which initially is in PE 2).

In parallel, each PE computes a shift value, which is simply the difference between its own index and the initial location. This shift value is shown in the third column in binary notation. The maximum shift value is 14, which is less than 2^4; thus, four passes through the flip network are required to spread the data.

The first pass is a PE-to-PE transfer with a shift-permutation of 8 places. The bit-slice with weight 8 of the shift value is used as a mask; where the bit is 0, the PE retains its stored value and where the bit is 1 the PE accepts data from the flip network. The fifth column of Figure 8 shows the values stored in each PE after this pass. PE's 0 through 10 are masked off and do not change state; PE's 11 through 18 accept data from PE's 3 through 10, respectively.

The second pass is a shift permutation of 4 places with the weight 4 bit-slice of the shift value used as a mask. PE's 6 through 10 and 15 through 18 accept data from PE's 2 through 6 and 11 through 14, respectively. The sixth column of Figure 8 shows the result.

Similarly, two more passes are executed with shifts of 2 places and 1 place, respectively, and with the weight 2 and weight 1 bit-slices of the shift value as masks, respectively. The last column of Figure 8 shows the result; this is pattern (2).

As long as the shift value bit-slices are treated in the correct order (most-significant bit-slice first), spreading can be performed

			DATA VALUE				
PE INDEX	INITIAL LOCATION OF DATA	SHIFT VALUE 8 4 2 1	INITIALLY	AFTER 8 SHIFT	AFTER 4 SHIFT	AFTER 2 SHIFT	AFTER 1 SHIFT
0	0	0 0 0 0	a	a	a	a	a
1	0	0 0 0 1	b	b	b	b	a
2	0	0 0 1 0	c	c	c	a	a
3	0	0 0 1 1	d	d	d	b	a
4	1	0 0 1 1	e	e	e	c	b
5	2	0 0 1 1	–	–	–	d	c
6	2	0 1 0 0	–	–	c	c	c
7	2	0 1 0 1	–	–	d	d	c
8	2	0 1 1 0	–	–	e	c	c
9	2	0 1 1 1	–	–	–	d	c
10	3	0 1 1 1	–	–	–	e	d
11	3	1 0 0 0	–	d	d	d	d
12	3	1 0 0 1	–	e	e	e	d
13	3	1 0 1 0	–	–	–	d	d
14	3	1 0 1 1	–	–	–	e	d
15	3	1 1 0 0	–	–	d	d	d
16	3	1 1 0 1	–	–	e	e	d
17	3	1 1 1 0	–	–	–	d	d
18	4	1 1 1 0	–	–	–	e	e

Figure 8. Irregular Spread Example

without collisions. Data can be spread into 2^n PE's with n passes or less if all shift values are non-negative.

Spreads with negative shift values require a modified method. First, all shift values are biased by a positive constant so that they are all non-negative. Then, certain bit-slices of the shift value field are complemented (the bit-slices corresponding to 1 bits in the bias constant). The result is a shift value where some bit-slices have negative weights and some have positive weights. The spread algorithm is then followed except that negative shifts are performed whenever negative-weight bit-slices are used as masks. The negative shifts are done with mirrors (with mirrored PE masks). If the bias constant is odd, the least-significant shift-value bit-slice has a negative weight and then an extra pass through the flip network is required to remirror the data into normal order. Data can be spread into 2^n PE's with n + 1 passes at most.

Irregular Compressing

Compressing (closing) takes data items from a scattered set of PE's and packs them into a contiguous set of PE's while preserving their relative order. It is the inverse operation of spreading and can be performed by reversing the steps of a spread.

Conclusions

The flip network scrambles and unscrambles data for the MDA memory. It also can perform the PE-to-PE routing required for many problems.

There is close connection between the flip network and the perfect shuffle. One can implement any flip network permutation with a few passes through a shuffle-exchange network. In many applications like the fast-Fourier-transform, a shuffle is used to pair up certain items. One pass through a flip network will also pair up the same items; the pairs may be ordered differently, however.

Irregular spreading and compressing can be performed in a few passes through the network. These operations are useful in image warping, rotation, magnification, and resampling.

References

1. K. E. Batcher, "The Multi-Dimensional-Access Memory in STARAN." 1975 Sagamore Computer Conference on Parallel Processing, p. 167; also submitted for publication in the IEEETC Special Issue on Parallel Processing.

2. L. H. Bauer, "Implementation of Data Manipulating Functions on the STARAN Associative Processor." 1974 Sagamore Computer Conference, pp 209-227.

3. H. S. Stone, "Parallel Processing with the Perfect Shuffle." IEETC Vol. C-20, pp. 153-161 (February 1971).

CONSTRUCTION OF A VERSATILE DATA MANIPULATOR FOR PARALLEL/ASSOCIATIVE PROCESSORS

W. W. Gaertner, M. P. Patel, C. T. Retter and I. M. Singh
W. W. Gaertner Research, Inc.
205 Saddle Hill Road
Stamford, Connecticut 06903

SUMMARY

At the 1973 Sagamore Computer Conference on Parallel Processing (Proceedings, p. 101) Tse-yun Feng proposed the design of a Versatile Data Manipulator for parallel/associative processors. It allows the programmer to establish a relationship between input and output words, such that any bit location in the input word may be specified as the data source for each of the bit locations in the output word. Both input and output data can be masked. The Rome Air Development Center has contracted with W. W. Gaertner Research, Inc. to perform the hardware design and construction of such a data manipulator to operate in conjunction with the STARAN computer at the RADCAP facility and the future Reconfigurable Computer System Design Facility (RCSDF) at RADC.

As shown in the block diagram of Figure 1, the data manipulator operates under the PIO Control of the STARAN computer, but could also be interfaced to other computers such as the QM-1 of the RCSDF. The contents of the input and output masks, of the Address Control Registers (ACR), and of the Input and Output Control Registers (ICR and OCR), as well as the data to be manipulated, are entered via the 256-bit wide PIO Buffer Interface. The manipulated data leave the data manipulator via the same interface. The instruction repertoire of the data manipulator allows one to load the various address registers and masks, and to start and stop data manipulation. Self-test is performed by loading address and input-data registers, allowing verification of correct operation even without assist from the STARAN computer. Details of physical construction and operation of the Data Manipulator are also presented.

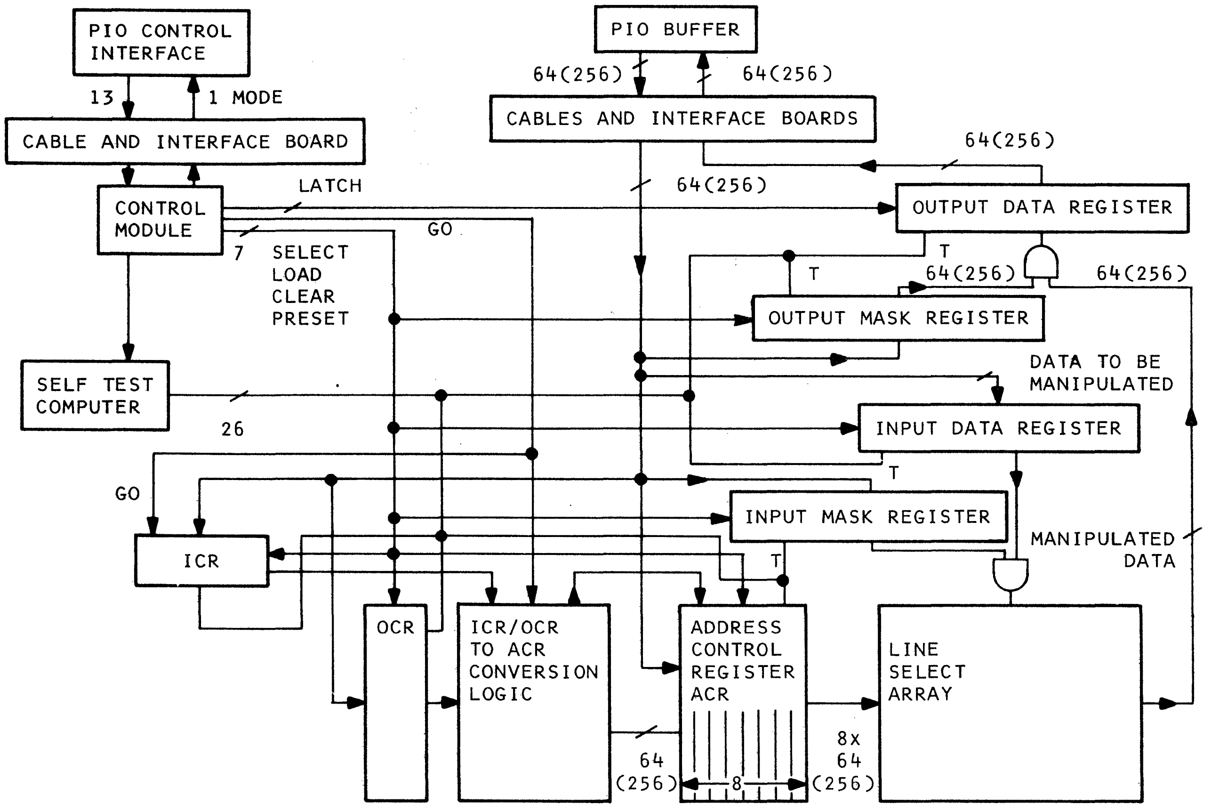


Figure 1 - Block Diagram of Data Manipulator

FAST ALGORITHMS FOR BOUNDING THE PERFORMANCE OF MULTIPROCESSOR SYSTEMS

Chao-Chih Yang
Department of Information Sciences
University of Alabama in Birmingham
University Station, Birmingham, Alabama 35294

Abstract--Given a finite set of partially ordered tasks with arbitrary execution times, more efficient methods for finding two types of sharper lower bounds in scheduling these tasks on a multiprocessor system are proposed. These bounds include a lower bound on the number of processors with the shortest execution time and that on the execution time under a specific number of available processors. This paper proposes fast algorithms for deriving two types of precedence partitions known as the earliest and the latest precedence partition, an equalization procedure for refining these partitions if their elements involve unequal time intervals, an algebraic method for recursively determining the lengths of all possible time intervals and a new technique for finding the numbers of common objects in these intervals. The determination of both types of sharper lower bounds follow the method proposed by Fernandez and Bussell. Working examples are used for illustrating all proposed concepts.

Introduction

Given a finite set of partially ordered tasks (referred to as a POSET), the scheduling of these tasks in a multiprocessor system has been widely studied [2], [7]. One of the important aspects of this scheduling problem is the determination of an optimal schedule where the optimality has been established with different objectives as done by McNaughton [15] and Hu [11] among others. Although the problem of searching for an optimal schedule for an arbitrary finite POSET could, in principle, be solved by performing a finite number of examinations, such an exhaustive searching is quite time consuming and becomes impractical for a relatively large POSET. Coffman and Graham [8] among others indicated that an efficient scheduling algorithm must be essentially nonenumerative or polynomial-time-bounded. Thus the enumerative searching method is inefficient since the number of steps involved is exponential in the number of tasks. Ullman [18] showed that the optimization problem of scheduling tasks in a POSET on m processors for all m is nondeterministic polynomial time complete. Hence, the problem of devising an optimal scheduling algorithm for an arbitrary POSET is quite difficult to be solved and there would be little hope that many more efficient algorithms in this aspect will be found. Consequently, a possible attitude toward the scheduling problem would be in devoting more effort to design algorithms such that the performance of the schedules induced by these algorithms is near optimal. Along this line, once a schedule is developed, its performance should be evaluated based on a bench mark as manifested in the experimental work done by Adam, et al. [1] and Kohler [13]. Therefore, efficient algorithms for bounding the performance of multiprocessor systems are of paramount importance and will be concerned in this paper.

In a computer system with m identical and independent processors, Baer and Estrin [3] suggested procedures to determine a lower and an upper bound on m required for maximum parallelism in a bilogic precedence graph. McNaughton [15], Hu [11], Chen and Epley [6], Barskiy [4], Ramamoorthy, et al. [17], Kraska [12] and Fernandez and Bussell [9] developed methods for finding lower bounds on m some among which are applicable only for special precedence graphs such as trees or those having tasks with equal execution times. Fernandez and Bussell also made an analytical study which showed that their lower bound on m is equivalent to Barskiy's result and is sharper than others. However, Barskiy's method is not practical for a POSET containing tasks with different execution times. The lower bounds on time for a specific m include those proposed by Hu [11] and Fernandez and Bussell [9] where the latter is also valid for any other arbitrary POSET. The methods for finding the upper bounds on the minimum number of processors were developed by Fulkerson [10], Barskiy [4], Ramamoorthy, et al. [17] and Fernandez and Bussell [9] among which the last one yields a sharper result. Although the bounds determined by the methods proposed by Fernandez and Bussell are sharper, the computation complexity of their methods limits their practical usefulness unless ways are found to improve the computation speed. This improvement will be considered in this paper.

In this paper, we shall furnish detailed algorithms for finding the earliest precedence partition (EPP) and the latest precedence partition (LPP) on a POSET containing tasks with different execution times, an equalization procedure for refining both of these partitions, an algebraic method for recursively determining the lengths of all possible time intervals and a new technique based on recursion, partition and successive reduction for finding the numbers of common objects in these intervals. The algorithms for finding those precedence partitions are fast, the equalization is worthwhile, particularly when some tasks in a POSET require much longer execution times than those of others and the method for finding the numbers of common objects does not rely on any load density function [9], does not use any bag [9] and does not perform any slower union operation the latter of which is replaced by much faster algebraic operations. Illustrations are also provided for demonstrating all new proposed concepts for a general case involving tasks with different execution times.

The case involving tasks with equal execution times is simpler than the general case and can be straightforwardly developed from the latter.

The Earliest and the Latest Precedence Partition

We consider a general case in which the EPP and the LPP are both on a given POSET containing tasks with different execution times. When a task is decomposed into a finite number of pieces, each such piece is called a subtask and assumes the same label as that of its decomposing task. This type of decomposition will be implicitly performed in the algorithms for finding the EPP or LPP.

Definition 1. An unexecuted task or subtask is a candidate at time t if all its predecessors have been completely executed by t. The candidate set at t is the set of all candidates at t.

Theorem 1. Every candidate set is independent.

The independence means that for every pair of distinct candidates x and y in a candidate set, x does not precede y and vice versa. This Theorem can easily be proved by contradiction. Note that a set of unexecuted objects being independent may not be a candidate set unless Definition 1 is satisfied.

When a POSET is executed, the overall cost depends on the availability of processors. Consider a case in which there is no processor constraint. Under this condition, the length of a schedule has the least possible value which is the length of a longest path in the precedence graph of the POSET. This path is known as a critical path.

Definition 2. Given a finite POSET S, let U_1 through U_k be some subsets (not necessarily disjoint or even distinct for every pair of these subsets) of S. If these subsets satisfy that 1) their union is equal to S, 2) each task or subtask corresponding to S appears in one and only one such subset, 3) each subset U_i for $i = 1,2,\ldots,k$ is independent, and 4) every pair of adjacent subsets U_i and U_{i+1} for $i = 1,\ldots,k-1$ in the sequence $U_1,U_2,\ldots,U_k$ has at least one precedence relation between two distinct labels or between two distinct subtasks with the same label, then the sequence is called a precedence partition (PP) on the POSET S.

Note that a PP on a POSET S is defined as a sequence of some subsets of S rather than a set of these subsets since condition 2) of this definition does not necessarily imply that these subsets of the PP are pairwise disjoint because of the possible existence of precedence constrained subtasks having the same label in some elements of the PP. Although these subtasks have the same label, they are distinct pieces decomposed from the task with that label and there are precedence relations among these subtasks with the same label so that condition 4) of this definition can hold. As will be seen in a later section entitled 'Equalization', a special type of precedence partitions may contain duplicated elements.

Definition 3. The latest possible time at which the execution of a task or subtask corresponding to a POSET S must be started without affecting the length of a schedule being longer than the length of a critical path in the precedence graph of the POSET S under the condition in which there is no processor constraint is called the latest starting time (LST) of the task or subtask. The sum of the LST of a task or subtask and its execution time is called the latest completion time (LCT) of the task or subtask. The sequence of some subsets of S with each such subset containing all tasks and/or subtasks having the same LST and with all such subsets arranged in the ascending order of those LST's is called the LPP on S.

Definition 4. The earliest possible time at which a task or subtask becomes a candidate is called the earliest starting time (EST) of the task or subtask when there is no processor constraint. Similar to Definition 3, we can define the earliest completion time (ECT) of a task or subtask and the EPP on a POSET.

Algorithms for Finding the EPP and the LPP

Definition 5. When the subtasks decomposed from a task have the same label as that of their decomposing task and this task has been partially executed, the remaining time required to finish the execution of this task is called the residual execution time of the task.

In the two algorithms to be proposed, a set S is assumed to be a nonempty and nonindependent POSET. Consequently, the detection of an empty POSET, an independent POSET and the existence of a circuit can be omitted.

Algorithm 1. An algorithm for finding the EPP on a POSET S containing n tasks with different execution times τ_x for $x = 1,\ldots,n$:

Step 1) Initialize $t := 0$ and $k_E := 0$ and store S, R, and τ_x for all x in S where R is the precedence relation on S (or the set of arcs in the precedence graph of the POSET S).

Step 2) Find the current set of successors by $R_2 := \{y|(x,y)\in R\}$.

Step 3) Find the current candidate set at t by $E_t := S - R_2$ and set $k_E := k_E + 1$.

Step 4) Print t, k_E and E_t.

Step 5) Find the least residual execution time of some objects in E_t by

$$\Delta t := \min_{\text{all } x \text{ in } E_t} \{\tau_x\},$$ set $t := t + \Delta t$ and update $\tau_x := \tau_x - \Delta t$ for each x in E_t.

Step 6) Update $S := S - \{x|x\in E_t \text{ and } \tau_x = 0\}$.

Step 7) Is $S = \phi$ where ϕ stands for the empty set? a) If so, set $t_m := t$, print t_m and the algorithm terminates. b) If not, update $R := R - \{(x,y) | x \in E_t \text{ and } \tau_x = 0\}$ and go to Step 2).

In this algorithm, the symbol E_t for each t denotes the candidate set at t as well as the element of the EPP containing tasks and/or subtasks with the same EST = t, the symbol t_m denotes the length of a critical path and k_E with positive integer values stands for the k_E-th element of the EPP.

For finding an LPP, we need to derive the inverse R^r of the precedence relation R on S by simply reversing the direction of every arc in the precedence graph of the POSET S.

Algorithm 2. An algorithm for finding the LPP on a POSET S containing n tasks with different execution times τ_x for $x = 1,2,\ldots,n$:

```
Step 1) Find $t_m$ by Algorithm 1.
Step 2) Construct the inverse of R by $R^r := \{(y,x) | (x,y) \in R\}$.
Step 3) Store S and $\tau_x$ for all x in S and initialize $t' := 0$ and $V := 1$.
Step 4) Find the current set of successors by $R_2 := \{x | (y,x) \in R^r\}$.
Step 5) Find the current candidate set by $D_{t'} := S - R_2$.
Step 6) Compute $\Delta t := \min_{\text{all } x \text{ in } D_{t'}} \{\tau_x\}$
    where $\tau_x$ is the residual execution time of x in $D_{t'}$, set $t' := t' + \Delta t$ and update $\tau_x := \tau_x - \Delta t$ for each x in $D_{t'}$.
Step 7) Update $S := S - \{x | x \in D_{t'} \text{ and } \tau_x = 0\}$.
Step 8) Is $S = \phi$? a) If so, set $L_0 := D_{t'}$ and $t := 0$, print t, V, and $L_0$ and the algorithm terminates. b) If not, update $R^r := R^r - \{(y,x) | y \in D_{t'} \text{ and } \tau_y = 0\}$, compute $t := t_m - t'$, set $L_t := D_{t'}$, print t, V and $L_t$, increment $V := V + 1$ and go to Step 4).
```

In this algorithm, the symbol L_t denotes the element of the LPP containing tasks and/or subtasks with the same LST = t and the values of V define the levels of tasks and/or subtasks as will be defined by the following definition. Let V_u be the maximum value of V. If we set

$$k_L = V_u - V + 1 \qquad (1)$$

then k_L with positive integer values stands for the k_L-th element of the LPP.

Note that Algorithms 1 and 2 can be also applied to a POSET containing tasks with equal execution times. However, they can be simplified for this special case.

Definition 6. If a task or subtask x is contained in the k_L-th element of an LPP, then the level of x, denoted by W(x), is defined by

$$W(x) = V_u - k_L + 1 \qquad (2a)$$

Theorem 2. The values of V derived from Algorithm 2 define the levels of the tasks and/or subtasks corresponding to a given POSET.

This Theorem is trivially true since (1) and (2a) imply that

$$W(x) = V \qquad (2b)$$

Note that when a task is decomposed, the level of the task itself is the highest one among those of its subtasks. Note also that the levels defined by Definition 6 are always positive integers and are not identical to those defined elsewhere [16] unless all tasks have equal execution times of 1 unit.

Illustration 1. Given a POSET $S_1 = \{1,2,3,4,5,6,7,8,9\}$ with the precedence relation $R = \{(1,2),(1,3),(2,4),(2,5),(3,6),(4,7),(4,8),(5,7),(5,8),(6,9),(7,9),(8,9)\}$, and the execution times $\tau_1 = \tau_3 = \tau_9 = 1$, $\tau_2 = 2$, $\tau_4 = \tau_5 = 20$, $\tau_6 = 12$, $\tau_7 = 40$ and $\tau_8 = 30$. As shown in Table I, the first row designated by t denotes the sequence of times from 0 through 64 where 64 equals the length t_m of a critical path, the next two rows designated by k_E and E_t denote the EPP and the following two rows designated by k_L and L_t denote the LPP. Although both k_E and k_L have the same maximum value 8 for this specific example, the numbers of elements in both of these partitions are, in general, not necessarily equal. The EPP has the EST sequence 0, 1, 2, 3, 14, 23, 53, 63 and the LPP has the LST sequence 0, 1, 3, 23, 33, 50, 51, 63 which are not identical. Note that tasks 2, 4, 5, 6, and 7 are each decomposed into 2 pieces in the EPP and tasks 7 and 8 are respectively decomposed into 4 and 3 pieces in the LPP.

An Equalization

Since the decomposition of a task x in the LPP may not be the same as that of the task x in the EPP, there does not exist a one-to-one correspondence from the subtasks decomposed from x in one PP onto those decomposed from the same task x in the other PP. This fact increases the complexity when the numbers of common objects as mentioned previously are computed. This requires that a one-to-one correspondence be established for each task having been decomposed in each PP. When the execution times of all tasks in a POSET are mutually commensurable [8], a trivial solution is firstly finding the greatest common divisor w of all such times, then decomposing every task x with $\tau_x = n_x w$ for $n_x > 1$ into n_x subtasks and finally deriving the EPP and the LPP on the refined precedence graph G_w [8], [16] which has vertices involving equal execution times of w units. However, this method may not be efficient because of involving quite a large number of partition elements, particularly when some tasks have their execution times being much longer than w. A feasible technique lies in providing a one-to-one and onto mapping such that the k-th subtask of task x in one PP can be mapped to the k'-th subtask of the same task x in the other PP if both subtasks have the same execution time but k and k' are not necessarily equal. The establishment of such one-to-one and onto mappings for all decomposing tasks is called an equalization. The equalized EPP and LPP on S are referred to as the refined EPP (REPP) and the refined LPP (RLPP) on S.

For obtaining the REPP and the RLPP, we need to augment some new elements by splitting some existing elements in either or both of the original EPP and LPP as generated by Algorithms 1 and 2 respectively in order to establish a one-to-one and onto mapping for each decomposing task. Each of these new elements is actually a duplication of some exiting element E_t or L_t and must be denoted by $E_{t'}$ or $L_{t'}$ for $t < t'$. The augmentation requires the following rules:

Rule 1) For each task or subtask x with $EST_x = LST_x = t_1$ having been decomposed into n_x subtasks in the interval or subinterval $[EST_x, ECT_x]$ of one PP at $t_2, t_3, \ldots,$ and t_{n_x}, the task or subtask x must be also identically decomposed in the same interval or subinterval of the other PP.

Rule 2) For each task y with $ECT_y < LST_y$ having been decomposed, we need to equalize the subintervals in both intervals $[EST_y, ECT_y]$ and $[LST_y, LCT_y]$ so that the i-th subtask y in the former interval and that in the latter interval have subintervals with the same length which is the execution time of i-th subtask with label y.

If a task z with $EST_z < LST_z < ECT_z < LCT_z$, we can consider three intervals $[LST_z, ECT_z]$, $[EST_z, LST_z]$ and $[ECT_z, LCT_z]$. For equalizing the subintervals of the first interval, we apply Rule 1. For equalizing those in the second and third intervals, we apply Rule 2. Note that some augmentation may induce new decomposing tasks which were not decomposed in the original EPP and/or LPP. Thus, an equalization needs a repeated refining and can be terminated when and only when all decomposing tasks including the induced ones have their one-to-one and onto mappings.

Illustration 2. Referring to the EPP and the LPP on the POSET S_1 as shown in Table I, tasks 2, 4, 5, 6, 7, and 8 are decomposed in either or both PP's without one-to-one correspondences. By Rule 1), we need to augment new elements $L_2 = \{2\}$, $L_{14} = \{4,5\}$, $E_{33} = E_{50} = E_{51} = \{7,8\}$ and $L_{53} = \{6,7,8\}$ in the interval [1,3] for task 2, in [3,23] for tasks 4 and 5, and in [23,63] for task 7. By Rule 2), we need to augment new elements $E_4 = \{4,5,6\}$ and $L_{52} = \{6,7,8\}$ in the intervals [2,14] and [51,63] for task 6. By Rule 1) again, we need to augment new elements $L_4 = \{4,5\}$ and $E_{52} = \{7,8\}$ in the subinterval [3,14] for tasks 4 and 5 and in the subinterval [51,53] for task 7. Up to this point, all tasks have one-to-one correspondences in both PP's so that the equalization is terminated. The REPP and the RLPP as obtained by the equalization above are shown by the rows designated by k, V, $L_0(k)$ and $E_0(k)$ in Table I. The values of k are from 1 through 13 which indicate the k-th elements. The values of V are also from 1 through 13 but in a reverse order. The latter values can stand for the levels of the tasks and subtasks in the elements of the RLPP. Thus, both Definition 6 and Theorem 2 with minor modifications are applicable for a RLPP. Let V_m be the maximum value of V. Then (2a) becomes

$$W(x) = V_m - k + 1 \qquad (2c)$$

Since there are 13 elements in either refined PP, there are 81 time intervals and also 81 numbers of common objects. On the other hand, if we use the EPP and the LPP on the refined graph G_1 containing vertices with equal execution times of w = 1 unit, there are 64 elements in either PP so that there are 2,080 numbers of common objects which is more than 25 times 81.

Determination of Time Intervals

Let $\Delta_1(k)$ for $k = 1,2,\ldots,V_m$ be the lengths of the time intervals spanned by the k-th elements of either refined PP on a POSET S.

Definition 7. The length $\Delta_j(k)$ for $j = 2,\ldots, V_m$ and $k = 1,2,\ldots,V_m - j + 1$ of the time intervals each spanned by j consecutive elements of either refined PP can be recursively defined as:

$$\Delta_2(k) = \sum_{i=k}^{k+1} \Delta_1(i), \text{ for } k = 1,\ldots,V_m - 1 \qquad (3a)$$

$$\Delta_j(k) = \sum_{i=k}^{k+1} \Delta_{j-1}(i) - \Delta_{j-2}(k+1), \text{ for}$$
$$j=3,4,\ldots,V_m \text{ and } k = 1,2,\ldots,V_m - j + 1 \qquad (3b)$$

If $\Delta_1(k) = w$ for all $k = 1,2,\ldots,V_u$, then

$$\Delta_j(k) = jw \qquad (3c)$$

for $j = 2,3,\ldots,V_u$ and $k = 1,2,\ldots,V_u - j + 1$.

It requires exactly $(V_m - 2)(V_m - 1)/2$ binary subtractions and exactly $(V_m - 1)V_m/2$ binary additions to compute the lengths of all time intervals by (3a) and (3b). In the special case it requires only $V_u - 1$ binary multiplications based on (3c).

Illustration 3. The lengths of time intervals each spanned by j consecutive elements of either refined PP are shown in Table II.

Determination of the Numbers of Common Objects

Let $E_0(k)$ and $L_0(k)$ for $k = 1,2,\ldots,V_m$ be the k-th elements of the REPP and the RLPP on an arbitrary POSET. Since each $E_0(k)$ is never empty and so is each $L_0(k)$ and every task or subtask with EST = LST must be contained in both of these sets, the intersection of $E_0(k)$ and $L_0(k)$ for each k is never empty. Each such intersection contains only common tasks and/or subtasks in the k-th elements of both refined PP's. Let

$$I_1(k) = E_0(k) \cap L_0(k) \text{ for } k = 1,\ldots,V_m \qquad (4a)$$

Then $I_1(k)$ is always a subset of $E_0(k)$ and that of $L_0(k)$ so that we can find the reduced sets from $E_0(k)$ and $L_0(k)$ by deleting those common objects in $I_1(k)$. Let these reduced sets be denoted by $E_1(k)$ and $L_1(k)$ for $k = 1,2,\ldots,V_m$. Then we can define

$$E_1(k) = \begin{cases} E_0(k) - I_1(k), & \text{for } k = 1,\ldots,V_m - 1 \\ \phi, & \text{otherwise} \end{cases} \quad (5a)$$

and

$$L_1(k) = \begin{cases} L_0(k) - I_1(k) & \text{for } k = 2,\ldots,V_m \\ \phi, & \text{otherwise} \end{cases} \quad (6a)$$

Since $I_1(k) \neq \phi$ for each k, the reduced sets $E_1(k)$ and $L_1(k)$ must be respectively some proper subsets of $E_0(k)$ and $L_0(k)$, i.e.,

$$E_1(k) \subset E_0(k) \quad (7a)$$

$$L_1(k) \subset L_0(k) \quad (8a)$$

and the intersection of $E_1(k)$ and $L_1(k)$ must be empty, i.e.,

$$E_1(k) \cap L_1(k) = \phi \quad (9a)$$

Now, we consider two consecutive reduced sets $E_1(k)$, $E_1(k + 1)$, $L_1(k)$ and $L_1(k + 1)$ for $k = 1,\ldots,V_m - 1$. Since $E_1(k) \cap L_1(k) = E_1(k + 1) \cap L_1(k + 1) = E_1(k + 1) \cap L_1(k) = \phi$ where the first two empty intersections are based on (9a) and the last one is due to the fact that there does not exist any object with LST < EST, we can compute the set of common objects in two consecutive reduced sets $E_1(k)$, $E_1(k + 1)$, $L_1(k)$ and $L_1(k + 1)$ as

$$I_2(k) = \begin{cases} E_1(k) \cap L_1(k + 1), & \text{for } k = 1,\ldots,V_m - 1 \\ \phi, & \text{otherwise} \end{cases} \quad (4b)$$

Then, $I_2(k)$ is always a subset of $E_1(k)$ and that of $L_1(k + 1)$ so that we can find $E_2(k)$ and $L_2(k)$ by deleting those common objects in $I_2(k)$ and $I_2(k - 1)$. Let these reduced sets be denoted by $E_2(k)$ and $L_2(k)$. Then we can define

$$E_2(k) = \begin{cases} E_1(k) - I_2(k), & \text{for } k = 1,\ldots,V_m - 2 \\ \phi, & \text{otherwise} \end{cases} \quad (5b)$$

and

$$L_2(k) = \begin{cases} L_1(k) - I_2(k - 1) & \text{for } k = 3,\ldots,V_m \\ \phi, & \text{otherwise} \end{cases} \quad (6b)$$

If we repeat the previous steps, we can define the set $I_j(k)$ of common objects in j consecutive reduced sets $E_{j-1}(k)$ through $E_{j-1}(k + j - 1)$ and $L_{j-1}(k)$ through $L_{j-1}(k + j - 1)$ and the further reduced sets $E_j(k)$ and $L_j(k)$ for $j = 3,\ldots,V_m$.

<u>Definition 8</u>. The sets $I_j(k)$, $E_j(k)$ and $L_j(k)$ for $j = 1,2,\ldots,V_m$ can be alternately and recursively defined as:

$$I_j(k) = \begin{cases} E_{j-1}(k) \cap L_{j-1}(k + j - 1), \\ \quad \text{for } k = 1,\ldots,V_m - j + 1 \\ \phi, \text{ otherwise} \end{cases} \quad (4c)$$

$$E_j(k) = \begin{cases} E_{j-1}(k) - I_j(k), & \text{for } k = 1,\ldots,V_m - j \\ \phi, & \text{otherwise} \end{cases} \quad (5c)$$

$$L_j(k) = \begin{cases} L_{j-1}(k) - I_j(k - j + 1), \text{ for} \\ \quad k = j + 1,\ldots,V_m \\ \phi, \text{ otherwise} \end{cases} \quad (6c)$$

Eqs. (4a), (5a) and (6a) coincide respectively with (4c), (5c) and (6c) for j = 1 and similarly, (4b), (5b) and (6b) coincide respectively with (4c), (5c) and (6c) for j = 2. There are at most $V_m(V_m + 1)/2$ nonempty sets $I_j(k)$ for $j = 1,\ldots,V_m$ and $k = 1,\ldots,V_m - j + 1$. The remaining sets $I_j(k)$ for $k = 2,\ldots,V_m$ and $j = V_m - k + 2,\ldots,V_m$ are always empty. There are at most $(V_m - 1)V_m/2$ nonempty reduced sets $E_j(k)$ for $j = 1,2,\ldots,V_m - 1$ and $k = 1,2,\ldots,V_m - j$. The remaining sets $E_j(k)$ for $k = 1,\ldots,V_m$ and $j = V_m - k + 1,\ldots,V_m$ are always empty. Similarly, there are at most $(V_m - 1)V_m/2$ nonempty reduced sets $L_j(k)$ for $j = 1,\ldots,V_m - 1$ and $k = j + 1,\ldots,V_m$. The remaining sets $L_j(k)$ for $j = 1,\ldots,V_m$ and $k = 1,\ldots,j$ are always empty. In summary, we have

$$I_j(k) = \phi \text{ for } k = 2,\ldots,V_m \text{ and } j = V_m - k + 2,\ldots,V_m \quad (4d)$$

$$E_j(k) = \phi \text{ for } k = 1,\ldots,V_m \text{ and } j = V_m - k + 1,\ldots,V_m \quad (5d)$$

and

$$L_j(k) = \phi \text{ for } j = 1,\ldots,V_m \text{ and } k = 1,\ldots,j \quad (6d)$$

It requires at most $V_m(V_m + 1)/2$ binary set intersection operations and at most $(V_m - 1)V_m$ binary set deletion operations to find all nonempty sets $I_j(k)$, $E_j(k)$ and $L_j(k)$.

Similar to (7a), (8a) and (9a), the successively reduced sets $E_j(k)$ and $L_j(k)$ for $j = 2,\ldots,V_m$ and $k = 1,2,\ldots,V_m$ have the following properties:

$$E_j(k) \subseteq E_{j-1}(k) \quad (7b)$$

$$L_j(k) \subseteq L_{j-1}(k) \quad (8b)$$

and

$$E_j(k) \cap L_j(k) = \phi \quad (9b)$$

Eqs. (7a), (7b), (5c), (8a), (8b) and (6c) imply that

$$I_j(k) \subseteq E_{j-1}(k) \quad (10a)$$

and

$$I_j(k) \subseteq L_{j-1}(k + j - 1) \quad (10b)$$

for $j = 1,\ldots,V_m$ and $k = 1,2,\ldots,V_m - j + 1$. The validity of (9b) can be proved by mathematical induction. For the inductive basis, it was shown that (9a) holds. Suppose that (9b) holds for all k and for some j > 1. Then by means of (5c) and (6c), we have

$$\begin{aligned} E_{j+1}(k) \cap L_{j+1}(k) &= (E_j(k) - I_{j+1}(k)) \cap \\ &\qquad (L_j(k) - I_{j+1}(k - j)) \\ &= E'_j(k) \cap L'_j(k) \\ &\subseteq E_j(k) \cap L_j(k) \end{aligned}$$

where $E'_j(k)$ and $L'_j(k)$ are respectively some subsets of $E_j(k)$ and $L_j(k)$ because of (10a) and (10b). By the inductive hypothesis, the intersection of $E_j(k)$ and $L_j(k)$ is empty so that (9b) holds for all $j = 1,2,\ldots,V_m$.

<u>Theorem 3</u>. The sets $I_j(k)$ for all $j = 1,\ldots,V_m$ and $k = 1,\ldots,V_m - j + 1$ form a partition on a given POSET S.

Proof. Firstly, we show that the union of all sets $I_j(k)$ for each k in $\{1,2,\ldots,V_m\}$ and for all $j = 1,2,\ldots,V_m - k + 1$ is equal to $E_0(k)$, i.e.

$$\bigcup_{j=1}^{V_m-k+1} I_j(k) = E_0(k) \text{ for } k = 1,\ldots,V_m$$

By means of (5c), we have

$$I_j(k) = E_{j-1}(k) - E_j(k) \qquad (4e)$$

Taking unions on both sides of (4e), we have

$$\bigcup_{j=1}^{V_m-k+1} I_j(k) = E_0(k) - E_{V_m-k+1}(k)$$

where $E_{V_m-k+1}(k)$ for $k = 1,\ldots,V_m$ are all empty as shown in (5d).

Secondly, the union of all (nonempty) sets $I_j(k)$ is equal to the given POSET S, i.e.,

$$\bigcup_{k=1}^{V_m} \bigcup_{j=1}^{V_m-k+1} I_j(k) = \bigcup_{k=1}^{V_m} E_0(k)$$

where the union on the right hand side is equal to S. Thus, condition 1) of Definition 2 is satisfied.

Thirdly, we show that each task or subtask is contained in exactly one set $I_j(k)$. Consider the intersection of two arbitrary nonempty sets $I_i(k)$ and $I_j(k)$ for some k and $i \neq j$. Let $\Delta = i - j \geq 1$. Then, by means of (4e), (7b) and the distributivity, we have

$$\begin{aligned} I_{j+\Delta}(k) \cap I_j(k) &= (E_{j+\Delta-1}(k) - E_{j+\Delta}(k)) \cap (E_{j-1}(k) - E_j(k)) \\ &= (E_{j+\Delta-1}(k) \cap E_{j-1}(k)) \\ &\quad - (E_{j+\Delta-1}(k) \cap E_j(k)) \\ &\quad - (E_{j+\Delta}(k) \cap E_{j-1}(k)) \\ &\quad + (E_{j+\Delta}(k) \cap E_j(k)) \end{aligned}$$

where the left two intersections are identical (to $E_{j+\Delta-1}(k)$) and are cancelled and so are the remaining two intersections (identical to $E_{j+\Delta}(k)$). Now, consider the intersection of two arbitrary nonempty sets $I_j(k)$ and $I_j(k')$ for some j and $k \neq k'$. This intersection contains either nothing or some subtasks, each of which denotes the same label of two distinct subtasks with precedence constraint. Thus, every task or subtask is contained in exactly one set $I_j(k)$ for some j and k and Condition 2) of Definition 2 is satisfied.

Since Conditions 1) and 2) of Definition 2 are both satisfied, the set of all nonempty sets $I_j(k)$ can be viewed as a partition on S.

Once $I_j(k)$ for all j and k are found, we can compute the numbers of common objects in all possible time intervals.

Definition 9. Let $W_j(k)$ be the numbers of common objects in j consecutive elements of both refined PP's. Then $W_j(k)$ can be recursively defined as:

$$W_1(k) = \Delta_1(k)\ \#(I_1(k)) \text{ for } k = 1,\ldots,V_m \qquad (11a)$$

$$W_2(k) = \Delta_1(k)\ \#(I_2(k)) + \sum_{i=k}^{k+1} W_1(i), \text{ for } k = 1,\ldots,V_m - 1 \qquad (11b)$$

$$W_j(k) = \Delta_1(k)\ \#(I_j(k)) + \sum_{i=k}^{k+1} W_{j-1}(i) - W_{j-2}(k+1), \text{ for } j = 3,4,\ldots,V_m \text{ and } k = 1,\ldots,V_m - j + 1 \qquad (11c)$$

where the symbol # applied to a set $I_j(k)$ indicates the number of objects in the set $I_j(k)$.

Finding $V_m(V_m+1)/2$ numbers $W_j(k)$ requires exactly $(V_m - 2)(V_m - 1)/2$ binary subtractions, at most $V_m(V_m + 1)/2$ binary multiplications and at most $(V_m - 1)V_m$ binary additions.

Illustration 4. For the refined PP's on S_1, the nonempty sets $E_j(k)$ and $I_{j+1}(k)$ are shown in Table III, the nonempty sets $L_j(k)$ and $I_{j+1}(k - j)$ are shown in Table IV, and the numbers $\Delta_1(k)\ \#(I_j(k))$ and $W_j(k)$ are shown in Table V. In these tables, every blank entry means that its corresponding set or number is empty or zero respectively.

Determination of Lower Bounds

Once the numbers $W_j(k)$ and the time intervals $\Delta_j(k)$ are available, a sharper lower bound m_{lb} on m and that t_{lb} on time can be derived by following the method of Fernandez and Bussell, i.e.,

$$m_{lb} = \max_{\text{all } j \text{ and } k} \{\lceil W_j(k)/\Delta_j(k) \rceil\} \qquad (12)$$

where $\lceil Z \rceil$ stands for the least integer greater than or equal to Z and

$$t_{lb} = \Delta_{V_m}(1) + \max_{\substack{\text{all } j \text{ and } k \\ W_j(k)/m > \Delta_j(k)}} \{W_j(k)/m - \Delta_j(k)\} \qquad (13)$$

where m is the number of available processors.

Since $W_j(k)/\Delta_j(k) \leq \lceil W_j(k)/\Delta_j(k) \rceil$, the lower bound t_{lb} on time for $m = m_{lb}$ is equal to $\Delta_{V_m}(1)$ which is the length of a critical path.

Illustration 5. By means of (12), the lower bound m_{lb} is equal to 3. By means of (13) based on m = 3, the lower bound t_{lb} is 64. If m = 2, the lower bound on time is 64.5.

Discussions and Conclusions

We have proposed two detailed algorithms for generating two types of precedence partitions on any POSET, an equalization of these partitions with elements involving unequal time intervals, a recursive and algebraic method for finding all time intervals and a faster method for finding $V_m(V_m + 1)/2$ common objects. Both types of precedence partitions are essentially needed for solving a scheduling bound problem. The algorithm for finding an LPP can induce "levels" as a by-product and the algorithm for generating an EPP can yield the length of a critical path as a by-product. Both of these by-products might be very useful in designing a level-oriented and critical path-oriented scheduling algorithms. In addition, a task being known as "essential" or "nonessential" can be defined by means of both precedence partitions in such a way that a task is essential if itself or its first subtask has EST = LST and is nonessential otherwise. All essential tasks are along critical paths. The price paid for an equalization to obtaining the refined precedence partitions might be worthwhile particularly in a case in which some tasks require much longer execution times as demonstrated in Illustration 2. The proposed method for finding common objects is based on the techniques of recursion, successive reduction and partition. However, the method of Fernandez and Bussell [9] relies on solving the following equations:

$$W_j(k) = \#\left(\bigcup_{i=k}^{k+j-1} L_0(i) \cap \bigcup_{i=k}^{k+j-1} E_0(i) \right) \qquad \text{(11d)}$$

for $j = 1,\dots,V_m$ and $k = 1,\dots,V_m - j + 1$. The intersections and unions based on (11d) involve set or even bag operands which are not only never empty but also gradually become larger as the computation proceeds whereas the intersections needed in our proposed method never involve bags but involve sets which gradually become smaller or even empty as j increases. In addition, no union operations are required in our proposed method since these slow operations are replaced by faster algebraic operations.

When a POSET contains tasks with different execution times in such a way that the number V_m of elements in either refined precedence partition is much less than t_m/w of the EPP or LPP corresponding to the refined precedence graph G_w, the proposed method is efficient. When a POSET containing tasks with equal execution times is relatively large, the proposed method is still applicable for this special case and is also worthwhile. The author tried twice to solve the Manacher's road map [14] (taken from [5]) with 86 tasks of the same execution time w = 1 by hand. When this computation was based on (11d), it was not only time consuming, but also difficult to avoid computation errors because the sets involved became more complicated as the computation proceeded. However, by means of our proposed method, it was much improved.

Acknowledgement

The work reported here was supported in part by N.I.H. Grant No. 5-501-RR05300-12 at the University of Alabama in Birmingham. The author thanks one referee who suggested a more specific title for this paper, Mrs. Janice Edwards for her efficient typing of the manuscript and Mr. Andrew W. Russo for implementing Algorithms 1 and 2 by PL/I programs.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," Comm. ACM, Vol. 17 (Dec., 1974), pp. 685-690.

[2] J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," ACM Computing Surveys, Vol. 5 (March, 1973), pp. 31-80.

[3] J. L. Baer and G. Estrin, "Bounds for Maximum Parallelism in a Bilogic Graph Model of Computations," IEEE Trans. on Computers, Vol. C-18 (Nov., 1969), pp. 1012-1014.

[4] A. B. Barskiy, "Minimizing the Number of Computing Devices Needed to Realize a Computational Process Within a Specified Time," Eng. Cybern. (USSR), No. 6 (1968), pp. 59-63.

[5] E. K. Bowdon, Sr., "Priority Assignment in a Network of Computers," IEEE Trans. on Computers, Vol. C-18 (Nov., 1969), pp. 1021-1026.

[6] Y. E. Chen and D. L. Epley, "Bounds on Memory Requirements of Multiprocessing Systems," Proc. 6th Annu. Allerton Conf., Circuit and Syst. Theory (1968), pp. 523-531.

[7] E. G. Coffman, Jr. and P. J. Denning, Operating Systems Theory, Prentice-Hall, (1973), pp. 83-143.

[8] E. G. Coffman, Jr. and R. L. Graham, "Optimal Scheduling for Two-Processor Systems," Acta Informatica, 1, (1972), pp. 200-213.

[9] E. B. Fernandez and B. Bussell, "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules," IEEE Trans. on Computers, Vol. C-22 (Aug., 1973), pp. 745-751.

[10] D. R. Fulkerson, "Scheduling in Project Networks," Proc. IBM Scientific Computing Symp. Combinatorial Problems (1966), pp. 72-92.

[11] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," Oper. Res., Vol. 9 (Nov., 1961), pp. 841-848.

[12] P. W. Kraska, "Parallelism Exploitation and Scheduling," Dept. Computer Sci., Univ. Ill., Urbana, Rep. UIUCDCS-R-62-518 (June, 1972).

[13] W. H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," IEEE Trans. on Computers, Vol. C-24 (Dec., 1975), pp. 1235-1238.

[14] G. K. Manacher, "Production and Stabilization of Real-Time Task Schedules," J. ACM, Vol. 14 (July, 1967), pp. 439-465.

[15] R. McNaughton, "Scheduling with Deadlines and Loss Functions," Management Sci., Vol. 6, (Oct., 1959), pp. 1-12.

[16] R. R. Muntz and E. G. Coffman, Jr., "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems," J. ACM, Vol. 17 (April, 1970), pp. 324-338. Also, "Optimal Preemptive Scheduling on Two-Processor Systems," IEEE Trans. on Computers, Vol. C-18 (Nov., 1969), pp. 1014-1020.

[17] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," IEEE Trans. on Computers, Vol. C-21 (Feb., 1972), pp. 137-146.

[18] J. D. Ullman, "Polynomial Complete Scheduling Problems," 4th Symposium on Operating Systems Principles, Yorktown Heights, New York (Oct., 1973), pp. 96-101.

Table I
Precedence Partitions on the POSET S_1

	t	0	1	2	3	4	14	23	33	50	51	52	53	63 64
EPP	k_E	1	2	3	4		5	6					7	8
	E_t	1	2 3	2 6	4 5 6		4 5	7 8					7	9
LPP	k_L	1	2		3			4	5	6	7			8
	L_t	1	2		4 5			7	7 8	3 7 8	6 7 8			9
k		1	2	3	4	5	6	7	8	9	10	11	12	13
V		13	12	11	10	9	8	7	6	5	4	3	2	1
RLPP	$L_0(k)$	1	2	2	4 5	4 5	4 5	7	7 8	3 7 8	6 7 8	6 7 8	6 7 8	9
REPP	$E_0(k)$	1	2 3	2 6	4 5 6	4 5 6	4 5	7 8	7 8	7 8	7 8	7 8	7	9

Table II

$\Delta_j(k)$ for $j = 1,\ldots,13$ and $k = 1,\ldots 14 - j$

j \ k	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	1	1	1	10	9	10	17	1	1	1	10	1
2	2	2	2	11	19	19	27	18	2	2	11	11	
3	3	3	12	20	29	36	28	19	3	12	12		
4	4	13	21	30	46	37	29	20	13	13			
5	14	22	31	47	47	38	30	30	14				
6	23	32	48	48	48	39	40	31					
7	33	49	49	49	49	49	41						
8	50	50	50	50	59	50							
9	51	51	51	60	60								
10	52	52	61	61									
11	53	62	62										
12	63	63											
13	64												

Table III

$E_j(k)$ and $I_{j+1}(k)$ for $j = 0,\ldots,12$ and $k = 1,\ldots,13 - j$

j	j+1	1		2		3		4		5		6		7		8		9		10		11		12		13	
0	1	1	1	2 3	2	2 6	2	4 5 6	4 5	4 5 6	4 5	4 5	4 5	7 8	7	7 8	7 8	7 8	7 8	7 8	7 8	7 8	7 8	7	7	9	9
1	2			3		6		6		6				8													
2	3			3		6		6		6				8													
3	4			3		6		6		6				8													
4	5			3		6		6		6				8													
5	6			3		6		6		6	6			8	8												
6	7			3		6		6																			
7	8			3	3	6		6	6																		
8	9					6																					
9	10					6	6																				
10	11																										
11	12																										
12	13																										

Table IV

$L_j(k)$ and $I_{j+1}(k-j)$ for $j = 0,\ldots,12$ and $k = j + 1,\ldots,13$

k																										j	j+1
1		2		3		4		5		6		7		8		9		10		11		12		13			
1	1	2	2	2	2	4 5	4 5	4 5	4 5	4 5	4 5	7	7	7 8	7 8	3 7 8	7 8	6 7 8	7 8	6 7 8	7 8	6 7 8	7	9	9	0	1
																3		6		6		6 8				1	2
																3		6		6		6 8				2	3
																3		6		6		6 8				3	4
																3		6		6		6 8				4	5
																3		6	6	6		6 8	8			5	6
																3				6		6				6	7
																3	3			6	6	6				7	8
																						6				8	9
																						6	6			9	10
																										10	11
																										11	12
																										12	13

Table V

$\Delta_1(k)\#(I_j(k))$ and $W_j(k)$ for $j = 1,\ldots,13$ and $k = 1,\ldots,14 - j$

j	k																									
	1		2		3		4		5		6		7		8		9		10		11		12		13	
1	1	1	1	1	1	1	2	2	20	20	18	18	10	10	34	34	2	2	2	2	2	2	10	10	1	1
2		2		2		3		22		38		28		44		36		4		4		12		11		
3		3		4		23		40		48		62		46		38		6		14		13				
4		5		24		41		50		82		64		48		40		16		15						
5		25		42		51		84		84		66		50		50		17								
6		43		52		85		86	10	96		68	10	70		51										
7		53		86		87		98		98		88		71												
8		87	1	89		99	1	101		118		89														
9		90		101		102		121		119																
10		102		104	1	123		122																		
11		105		125		124																				
12		126		126																						
13		127																								

TWO MODELS OF TASK OVERLAP WITHIN JOBS OF MULTIPROCESSING MULTIPROGRAMMING SYSTEMS

Mamoru Maekawa[a] and Donald L. Boyd
Department of Computer, Information and Control Sciences
University of Minnesota
Minneapolis, Minnesota 55455

Abstract -- The effects of task overlap within jobs on the job throughput rate in multiprocessing multiprogramming systems are studied. A job model which represents overlapping tasks within individual jobs is constructed and then extended so as to represent a system model. The system model is further refined so as to include two different CPU scheduling algorithms; nonpreemptive and processor-sharing. Having a high degree of overlap within a job appears to be equivalent to the addition of another job in the system when nonpreemptive algorithms are used. When a processor-sharing algorithm is used, a high degree of overlap within a job is approximately equivalent to doubling the number of jobs in the system.

1. Introduction

Since the introduction, in the early 1960's, of independent input/output (I/O) channels to remedy the disparity in speed between CPU computations and I/O device operations, concurrent tasks have become a common feature of almost all computer systems. Concurrency of operations, CPU tasks and I/O tasks, may occur between jobs of a multiprogramming system and within each of the individual jobs of a system. Although a great deal of analysis has been performed concerning the first type of concurrency [1, 3] only recently has attention been focused on internal concurrency within jobs as well as between jobs [9]. The complexity of analysis partially explains this deficit.

In [9], we modeled a batch-processing multiprogramming system in which the job model allowed concurrent CPU tasks and I/O tasks within jobs. We defined the degree of multiprogramming to be a two component measure of the number of concurrent jobs competing for processors and the potential internal concurrency of CPU tasks and I/O tasks within jobs. By using job throughput as a criterion for comparison, we were able to show, with the aid of numerical examples, the effects of both types of concurrency on throughput for three different system models. That is, up to a certain point, dependent upon the system model, concurrency within jobs has the same effect on throughput as did the addition of jobs with no internal concurrency. Thus improved throughput is achieved without the necessity of additional resources such as memory and with lower system overhead.

The degree of improvement appears to be a function of the computer system model, the variables include scheduling algorithms and service time distributions. The three system models used were:

1. A uniprogramming system allowing a single job system access under general distribution assumptions.
2. A multiprogramming system with competing tasks, but no processor preemption allowed. I/O service time was exponential while CPU service time followed a general distribution.
3. A multiprogramming system with a time-slicing algorithm used to allocate CPU time among competing tasks. In this model, CPU service time was represented by hyperexponential distribution.

The purpose of this paper is to present the analysis of two additional system models in order to further observe, through numerical examples, the effects of internal concurrency on job throughput rate. The two system models are similar to the last two models mentioned above. However in the present models, we will assume multiple identical CPUs, CPU service time to be governed by a hyperexponential distribution, and I/O service time to be governed by an Erlang distribution. The distributions of CPU and I/O service times are reported to be hyperexponential and Erlang [2, 7]. In the first model the processors are assumed to be nonpreemptive while in the second model the scheduling of CPU service follows a processor-sharing algorithm.

In the following section because of notational necessity we will repeat the fundamental assumptions for the basic job and system models. We will again define the measures to be used for numerical observations. In section three we will discuss the development of the equilibrium equations for the two system models mentioned above. In section four, we will describe some numerical examples, and conclude with some overall observations.

(a) Present address: Toshiba Research and Development Center, 1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki, 210, Japan

2. Assumptions and Notation

2.1 Job and System Models

The job model must conveniently represent concurrency of CPU and I/O tasks. We assume that a job consists of a series of alternating CPU and I/O tasks, beginning and ending with a CPU task. Concurrency is achieved when an I/O task is created before the completion of its corresponding CPU task. The next CPU task does not begin until after the I/O task is completed and the previous CPU task completes by issuing a system request (denoted as a wait) to synchronize the two tasks. This behavior is shown in Figure 2.1.

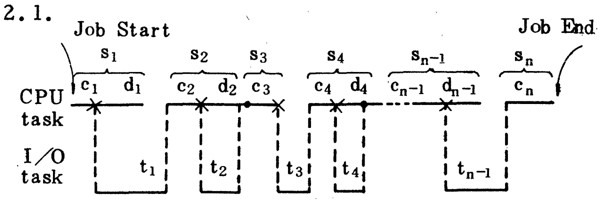


Figure 2.1 Job Behavior: x denotes I/O task creation and • denotes the wait request.

The notations and assumptions basic to the job model are the following:

1. A CPU task is defined as the processing time required of the CPU between two consecutive wait requests or the processing time between the beginning of the job and its first wait request or the processing time between the last wait request and the termination of the job. These times will be governed by the random variables s_i, $i = 1,2,...,n-1$, as shown in Figure 2.1. We assume that the s_i are mutually independent and identically distributed. Thus the CPU task times are denoted by the single random variable S with mean $1/\mu$ and distributed as a K phase hyperexponential with distribution function
$$F(s) = \sum_{k=1}^{K} w_k[1 - e^{-\mu_k s}]$$
where
$$\sum_{k=1}^{K} w_k = 1, \quad 0 \le w_k \le 1$$
and
$$\sum_{k=1}^{K} \frac{w_k}{\mu_k} = \frac{1}{\mu} \quad .$$
2. The random variables c_i, $i = 1,2,\cdots,n$, govern the processing time required of the CPU by a job from the beginning of execution or a wait request to the next issuance of an I/O request or to the termination of the job as shown in Figure 2.1. The c_i's are also assumed to be mutually independent and identically distributed, thus denoted by the single variable C. Furthermore, C is defined by two mutually independent random variables S and B where $C = \min\{S, B\}$. The random variable B is assumed to be exponentially distributed with mean $1/\nu$ and distribution function
$$H(b) = 1 - e^{-\nu b} \quad .$$
The variable B governs the time until an I/O request is made during a CPU task if overlap exists. B will be used in the measurement of overlap.
3. Denote by d_i the processing time required by the CPU from the issuance of an I/O request to its corresponding wait request. Thus d_i, $i = 1,2,\cdots,n-1$, is defined as $d_i = s_i - c_i$ and denoted by the single random variable D.
4. Denote by t_i, $i = 1,2,\cdots,n-1$, the i-th I/O channel time requirement. Assume that the t_i's are mutually independent and identically distributed and thus denoted by the single variable T with mean $1/\lambda$, and distributed as an Erlang of order U with distribution function
$$G(t) = \int_0^t \frac{(\lambda U)(\lambda U x)^{U-1} e^{-\lambda U x}}{(U-1)!}\, dx$$
5. The three random variables S, B and T are assumed to be statistically independent.
6. After each partial CPU task of length c_i, a job leaves the system with a fixed probability q or issues an I/O request with probability 1-q. Therefore the number of CPU tasks of a job is geometrically distributed with mean $1/q$, $0 < q \le 1$.

With these assumptions and the notation presented above, the behavior of a job as it flows through the system may be represented as a state transition graph.

The individual states are defined as follows:

A_1: A CPU task has been requested but its corresponding I/O processor task has not yet been requested.

A_2: Both a CPU and I/O processor task have been requested and neither of them as yet have completed.

A_3: The I/O processor task has completed but the corresponding CPU task is not yet completed.

A_4: The CPU task has completed but the corresponding I/O processor task is not yet completed.

Thus a job enters the system at state A_1 and cycles through the system and finally departs from the system after an average number of $1/q$ visits to state A_1. The directed cycles are shown in Figure 2.2.

Using this job model, the batch-processing multiprogramming system is now modeled by the number of jobs in each of the job states A_i, $i = 1, 2,3,4$. We assume that the system consists of M identical CPU's and N identical I/O channels. By assumption, a departing job is immediately replaced by a new job, thus a constant number L jobs remain present in the system at all times. The closed, cyclic queuing model representing the basic system model is depicted in Figure 2.2.

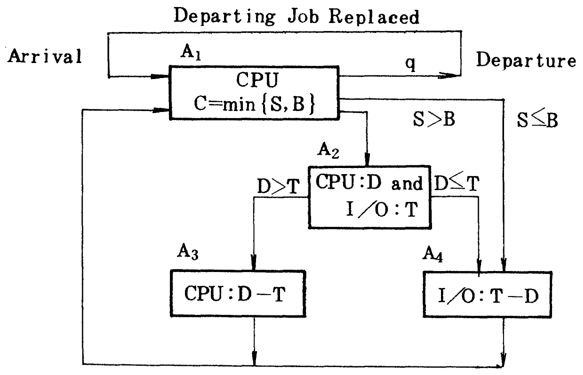


Figure 2.2 System Model

2.2 Measure — Degree of Multiprogramming and Job Throughput Rate

As shown in Figure 2.2 concurrency of CPU and I/O tasks within a job is achieved when the job enters state A_2. The probability of entering state A_2 is given by $Pr[S > B]$ and denoted by ω. ω will be called the overlap ratio and is an indication of the portion of CPU task time which can be potentially overlapped with an I/O task. It is shown in [9] that as the mean of B becomes large (no overlap), ω approaches zero, and as the mean of B becomes small (perfect overlap), ω approaches one.

The degree of overlap between different jobs in a multiprogramming system is usually measured by the number of jobs in the system per CPU or per I/O processor. Denote by Ω the number of jobs per CPU. The vector (Ω, ω) will be used to measure overlap for the entire system and is defined to be the degree of multiprogramming. The job throughput rate of the system might be expected to increase as Ω or ω increases. This rate will be calculated for different values of (Ω, ω).

The job throughput rate of the system, R, is defined in terms of the CPU utilization, Pr [CPU busy], and the total CPU time required by an average job in the system, T_c. Thus

$$R = \frac{M}{T_c} Pr[\text{CPU busy}]$$

$$= \frac{M}{(1/q-1)1/\mu + E[C]} Pr[\text{CPU busy}]$$

Similarly, R is related to the I/O processor utilization, Pr[I/O processor busy] and the total I/O processor time required by an average job in the system, T_I, as follows

$$R = \frac{N}{T_I} Pr[\text{I/O processor busy}]$$

$$= \frac{N}{(1/q-1)1/\lambda} Pr[\text{I/O processor busy}]$$

Define $R(M,N,\Omega,\omega)$ to be the job throughput rate of a system with M CPU's, N I/O channels, and a degree of multiprogramming (Ω,ω). R is clearly bounded by

$$R(M,N,1/M,0) \le R \le R(M,N,\infty,1)$$

It is shown in [9] that under general distribution assumptions these limits are given by

$$q\frac{\mu\lambda}{\lambda+(1-q)\mu} \le R \le \min\left\{M\mu\frac{q}{1-q},\ N\lambda\frac{q}{1-q}\right\}.$$

Improvement of the job throughput rate $\frac{R(M,N,\Omega,\omega)}{R(M,N,1/M,0)}$ is bounded by

$$1 \le \frac{R(M,N,\Omega,\omega)}{R(M,N,1/M,0)} \le \min\left\{\frac{M}{1-q}\left(1+(1-q)\frac{\mu}{\lambda}\right), \frac{N}{1-q}\left(\frac{\lambda}{\mu}+(1-q)\right)\right\}.$$

The upper bound takes the maximum when $M\mu = N\lambda$, thus given by

$$M/(1-q) + N \quad .$$

Two detailed system models are outlined in the following section.

3. Two Specific Cases of the System Model

Further analysis of the general model discussed in section two which would lead to useful results appears to be fruitless. In this section, we will concentrate on the algorithms for scheduling CPU and I/O tasks. We will assume that CPU and I/O tasks times are distributed as hyperexponential and Erlang respectively. Further, it is assumed that once an I/O task is in the running state, it is not subject to preemptions. The system consists of M identical CPU's, N identical I/O channels, and will support a fixed number of jobs, L, at all times.

The two cases of the system model will be distinguished by their assumptions on the scheduling of CPU tasks. However a few remarks are necessary concerning the scheduling of I/O channels to I/O tasks. Note that new I/O tasks may exist in states A_2 or A_4 as shown in Figure 2.2. The question arises as to what selection algorithm would best enhance the job throughput rate of the system. In [9], it was argued that the following, SCHEME A, is at least as good as any other strategy because it maintains a high arrival rate of new CPU tasks.

SCHEME A: Tasks at state A_4 are given a higher selection priority than those at state A_2. Among tasks at the same state, a task is randomly selected.

Although the analysis in this section assumes SCHEME A, the following SCHEME B has also been considered and included in the numerical examples of the following section.

SCHEME B: Tasks at state A_2 are given a higher selection priority than those at state A_4. Among tasks at the same state, a task is randomly selected.

3.1 Case 1 - Nonpreemption

In this model we will assume that after a CPU task has acquired a CPU it will complete CPU service without interruptions. Since CPU tasks are statistically independent and identically distributed and since preemptions are not allowed, the queue selection algorithm will have no effect on the job throughput rate of the system. Thus any selection algorithm such as a random or FCFS may be assumed.

This analysis utilizes the method of states due to Erlang [4] for I/O time. The hyperexponential distribution for CPU time is also analyzed using a similar method. The distribution function of the hyperexponential distribution is a weighted sum of K exponentials, thus CPU time can be simulated by a set of K CPU phases. When a CPU task is allowed access to a CPU, it chooses phase i with probability w_i. Phase i is exponentially distributed with mean $1/\mu_i$. Due to the severe limitation on the space allowed we eliminate the detailed analysis and refer interested readers to [10].

3.2 Case 2 - Processor-Sharing

In this model we will use a processor-sharing algorithm [8] to assign tasks to the CPU's. Note that although this algorithm is not realizable, it may be viewed as a limiting case of the classical round-robin algorithm where the time quantum size approaches zero.

It is assumed that a single CPU task may never occupy more than single CPU at any time. If the number of ready CPU tasks, c, is less than or equal to the number of CPU's, M, the rate of CPU processing assigned to each task is μ, otherwise the rate is reduced to $(M/c)\mu$. Therefore, the rate at which CPU tasks complete processing is $\min\{M\mu/c, \mu\}$. Again we exclude the detailed analysis and refer interested readers to [10].

4. Numerical Examples

We will use five example job sets to demonstrate the relationship of the degree of multiprogramming on the job throughput rate of the system for the models discussed in section three. In each of the five sets, the mean CPU time, $1/\mu$, and the probability that a job leaves the system, q, are held constant at $1/\mu$ equal to 1 and q equal to 0.02. The queuing algorithm for I/O processors is SCHEME A except in cases where the effect of I/O processor queuing is observed. The random variable B which primarily determines the duration from the beginning of a CPU task to the issuance of its corresponding I/O request is exponentially distributed with mean $1/\nu$. The parameter ν is varied from 0 to a very large number thus causing the overlap ratio, ω, to vary from 0 to 1. Other parameters such as the balance of the system, λ/μ, the number of CPU's, M, the number of I/O channels, N, and queuing disciplines for the CPU's will be varied. The characteristics of the five job sets are displayed in Table 4.1.

Numerical results are graphically presented. Each figure is a plot of improvement of the job throughput rate and the degree of multiprogramming, (Ω, ω). The improvement of the job throughput rate is defined as

$$\left(\frac{R(M,N,\Omega,\omega)}{R(M,N,1/M,O)} - 1\right) \times 100$$

That is, the improvement is measured as the percentage of increase in job throughput rate over a system with a single job which has no overlap within a job. (Ω,ω) is represented as a linear function $(\Omega + \omega)$, where $\Omega = 1,2,3,\cdots$ and $0 \leq \omega \leq 1$.

The plots displayed in Figures 4.1 and 4.2 are for job sets A and C respectively under a balanced system. That is, we assume that a system is balanced when the average job's demand for the CPU is equal to its demand for the I/O channel, thus λ/μ is equal to (1-q). Three observations are made. First, an increase in the job throughput rate is obtained by increasing the overlap ratio. In fact, under nonpreemptive schedules the following approximation can be observed for both job sets A and C.

$$R(1,1,\Omega,1) \approx R(1,1,\Omega+1, 0) \qquad (4.1)$$

This relation was also observed in [9]. That is, a high overlap ratio appears to be equivalent to

Table 4.1 Example Job Sets[(b)]

Job Set Identifier	CPU time distribution	I/O time distribution
A	Exponential	Exponential
B	Hyperexponential (Variance -4)	Exponential
C	Hyperexponential (Variance -16)	Exponential
D	Exponential	Erlang
E	Hyperexponential (Variance -16)	Erlang

(b) All hyperexponential distributions are two phases and all Erlang distributions are two stages.

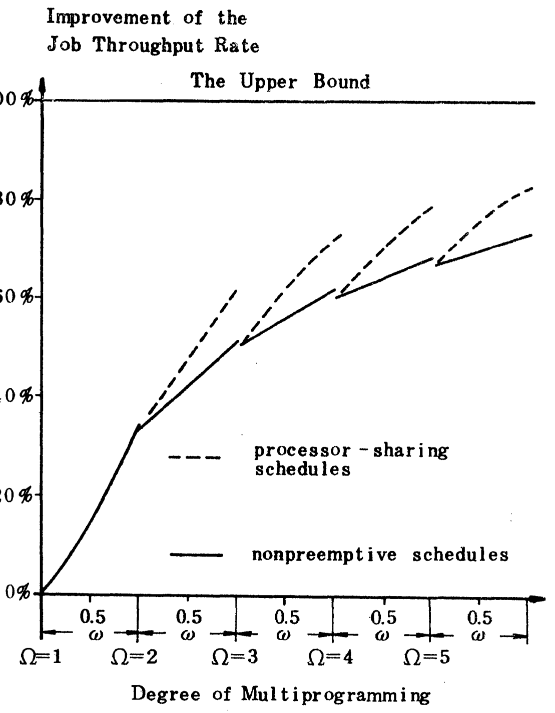


Figure 4.1 Improvement of the Job Throughput Rate for Job Set A in a Balanced System.

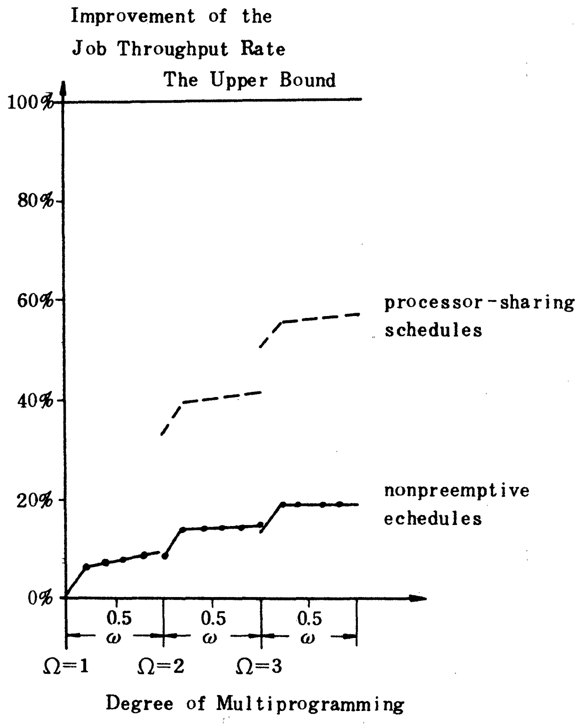


Figure 4.2 Improvement of the Job Throughput Rate for Job Set C in a Balanced System.

the addition of another job in the system. This relation holds only for systems with a single CPU and a single I/O processor.

The improvement of the job throughput rate is greater under preemptive schedule than under nonpreemptive schedules. Under processor-sharing schedules, which is the limiting case of the preemptive schedule the following approximation is observed for job set A.

$$R(1,1,\Omega,\omega) \approx R(1,1,\Omega(1+\omega),0) \quad . \qquad (4.2)$$

That is, having a high overlap ratio appears to be equivalent to doubling the number of jobs in the system. Relation 4.2 shows that overlap within a job has more effect on the job throughput rate under processor-sharing schedules than under nonpreemptive schedules. As seen in Figure 4.2, Relation 4.2 seems to hold for only small values of ω when using job set C where the CPU time is hyperexponentially distributed with a high variance.

The second observation relates to the effect of time-slicing. In Figure 4.1, it is observed that, when there is some overlap within a job, the job throughput rate is increased by time-slicing even if the distribution of the CPU time is exponential. This phenomenon is interesting because, if the distribution of the CPU time is exponential, and if there is no overlap within a job, then time-slicing does not affect the job throughput rate due to the memoryless property of the exponential distribution. Thus the inclusion of overlap within a job causes time-slicing to have a significant effect on the job throughput rate.

Increase in the job throughput rate by time-slicing is explained as follows. Task overlap within a job can actually be obtained when the job is at state A_2. The number of jobs at state A_2 is limited by the number of CPU's. Multiplexing due to time-slicing creates an image of multiple CPU's and allows more jobs to stay at state A_2 at the same time. Hence, more jobs can obtain task overlap at the same time.

When the distribution of CPU time is hyperexponential, the improvement of the job throughput rate due to time-slicing is larger than when the distribution of CPU time is exponential. This is observed in Figure 4.2. Baskett [1] has shown that the throughput behavior of the system with hyperexponential CPU times under processor-sharing schedules is identical to the throughput behavior of the system with exponential CPU times with the same mean. Comparing Figure 4.1 with Figure 4.2 it is seen that Baskett's statement is true only when the overlap ratio is zero.

The third observation concerns job set C only. In Figure 4.2, it is observed that the job throughput rate improves for a small value of ω, but remains nearly constant for any further increases of ω. Thus it appears that even a small amount of overlap within a job will have an equivalent effect to total overlap within a job if the CPU time is governed by a hyperexponential distribution with a high variance.

Thus far observations have been made for job set A and job set C. The throughput behavior of job set B, as might be expected, is between the throughput behaviors of job set A and job set C. This is observed, for example, in Figure 4.3, where the throughput behaviors are shown under the nonpreemptive schedule. Apparently the coefficient of variation is one of the major factors which determine the throughput behavior. The coefficients of variation of job sets A, B and C are 1, 2, and 4, respectively.

Since the behavior of job set B is easily inferred from that of job sets A and C, we will not include this set in what follows.

The effect on job throughput due to changes of the I/O time distribution is observed in Figures 4.3 and 4.4. The system is balanced and has a single CPU and a single I/O channel. Observe that Erlang I/O time provides a higher job throughput rate than exponential I/O time. But when CPU time is hyperexponentially distributed with a high variance, the effect of I/O time distribution is very small. Relation 4.1 for nonpreemptive schedules still appears to hold for Erlang I/O time. Relation 4.2 for exponential CPU time under processor-sharing schedules also appears to hold for Erlang I/O time.

The effects on job throughput rate due to changes of system configurations are displayed in Figure 4.5. We assumed that a job requires on the average four times more I/O channel time than CPU time in order to maintain a balanced system. When the number of jobs increases, there are jumps in the job throughput rate up to a point at which the number of jobs is equal to the number of I/O channels. The height of the jumps decreases as the number of jobs increases. The following form of Relation 4.1 appears to hold.

$$R(1,N,\Omega,1) \approx R(1,N,\Omega+1, 0)$$

for $L \geqq N$. $R(1,N,\Omega,1)$ tends to be larger than $R(1,N,\Omega+1,0)$. If the number of jobs is less than the number of I/O channels, $R(1,N,L,0)$ is roughly estimated by

$$R(1,N,L,0) \approx \frac{L\lambda/(\mu+L\lambda)}{\lambda/(\mu+\lambda)} R(1,1,L,0)$$

$$= \frac{L(\mu+\lambda)}{\mu+L\lambda} R(1,1,L,0)$$

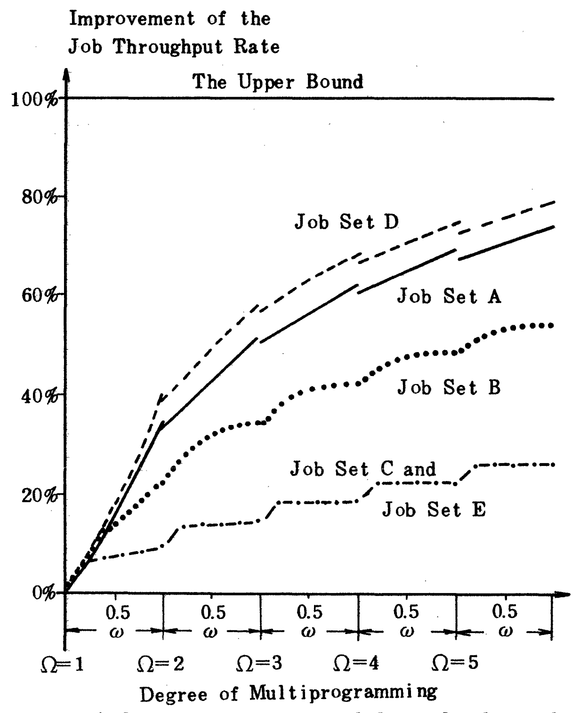


Figure 4.3 Improvement of the Job Throughput Rate in a Balanced System under Nonpreemptive Schedules.

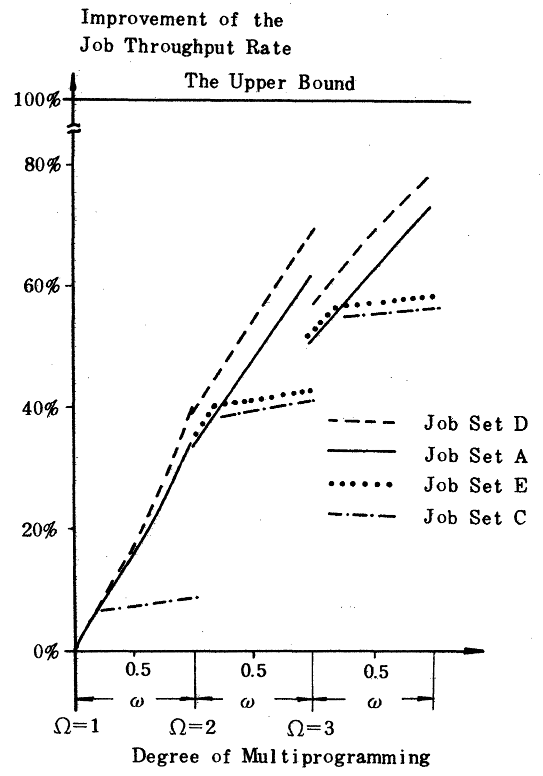


Figure 4.4 Improvement of the Job Throughput Rate for Job Sets A, C, D and E, in a Balanced System under Processor-Sharing Schedules.

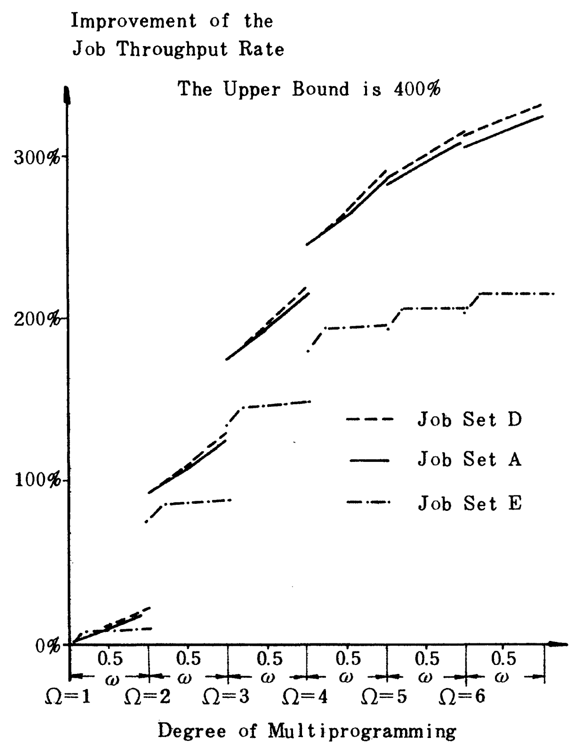


Figure 4.5 Improvement of the Job Throughput Rate in a System with a Single CPU and Four I/O Channels under Nonpreemptive Schedules.

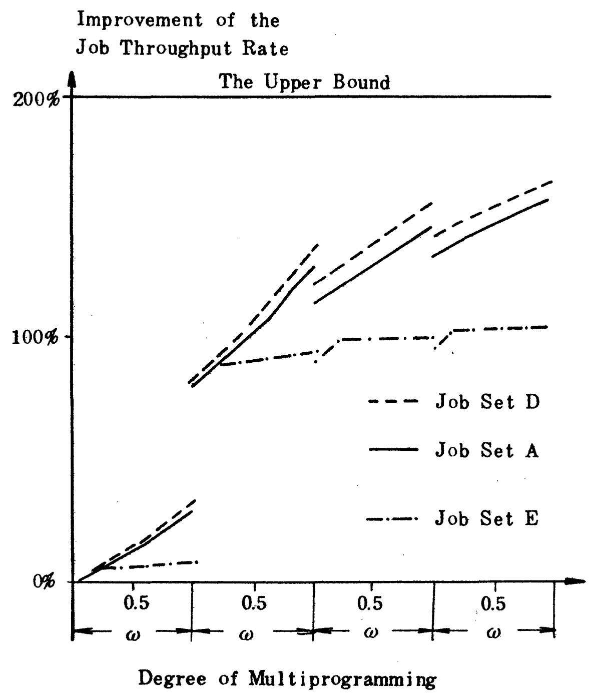


Figure 4.6 Improvement of the Job Throughput Rate in a System with Two CPU's and a Single I/O Channel Under Nonpreemptive Schedules.

Figure 4.6 is a plot of the improvement of the job throughput rate for a system with two CPU's and a single I/O channel. A job requires on the average twice as much CPU time as I/O channel time in order to maintain a balanced system. Except for the initial jump in the job throughput rate, the following general formula of Relation 4.1 appears to hold.

$$R(M,1,\Omega,1) \approx R(M,1,\Omega+1,0) \ .$$

Having a high overlap ratio appears to be equivalent to increasing Ω by 1; that is, equivalent to the addition of M jobs to the system.

Table 4.2 shows job throughput rate per CPU for job set D in two different system configurations. Both the systems are balanced. Queuing disciplines for the CPU's are nonpreemptive. It is observed that the job throughput rate per CPU is higher in the multiple processor system than the single processor system.

In Figure 4.7, job set D is plotted for a balanced system with two CPU's and two I/O processors under nonpreemptive schedules for the CPU's. Observe that the natural extension for multiple CPU's and I/O processors of Relation 4.3 and Relation 4.4 does not appear to hold. $R(M,N,\Omega,1)$ is smaller than $R(M,N,\Omega+1, 0)$. The difference between $R(M,N,\Omega,1)$ and $R(M,N,\Omega+1, 0)$ is attributed to the difference in the job throughput rate per CPU shown in Table 4.2 even for the same degree of multiprogramming.

The effect on job throughput rate due to the two queuing disciplines for the I/O processors, SCHEME A and SCHEME B, is observed next. Table 4.3 shows the improvement of the job throughput rate under SCHEME A and SCHEME B for job sets A, D, and E. The system has two CPU's and a single I/O channel. A multiple CPU system was chosen since queuing disciplines for the I/O channels do not affect the job throughput rate of a single CPU system. We assume that a job requires on the average twice as much CPU time as I/O channel time to maintain a balanced system. Queuing disciplines for the CPU's are nonpreemptive. Note that queuing disciplines for the I/O channels have little effect on the job throughput rate.

Table 4.2 Job Throughput Rate per CPU

	Degree of Multiprogramming	
	(2,0)	(2,1)
System with a Single CPU and a Single I/O Processor	0.0139	0.0158
System with Two CPU's and Two I/O Processors	0.0153	0.0173

(Ω, ω)	Job Set A		Job Set D		Job Set E	
	SCHEME A	SCHEME B	SCHEME A	SCHEME B	SCHEME A	SCHEME B
$(0.5,\omega)$ and $(1,\omega)$	No Difference		No Difference		No Difference	
(1.5, 0)	186.1	186.1	195.5	195.5	154.4	154.5
(1.5, 1)	229.0	227.2	239.6	237.4	166.8	165.5
(2.0)	211.6	211.6	222.2	222.2	161.1	161.1
(2, 0.9)	239.6	237.9	249.8	247.7	172.7	171.1

Table 4.3 Comparison of SCHEME A and SCHEME B

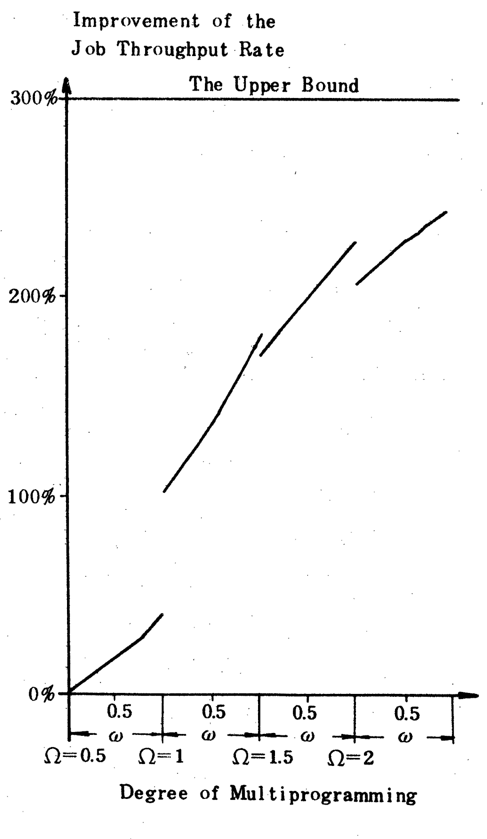


Figure 4.7 Improvement of the Job Thoughput Rate in a System with Two CPU's and Two I/O Processors under Nonpreemptive Schedules.

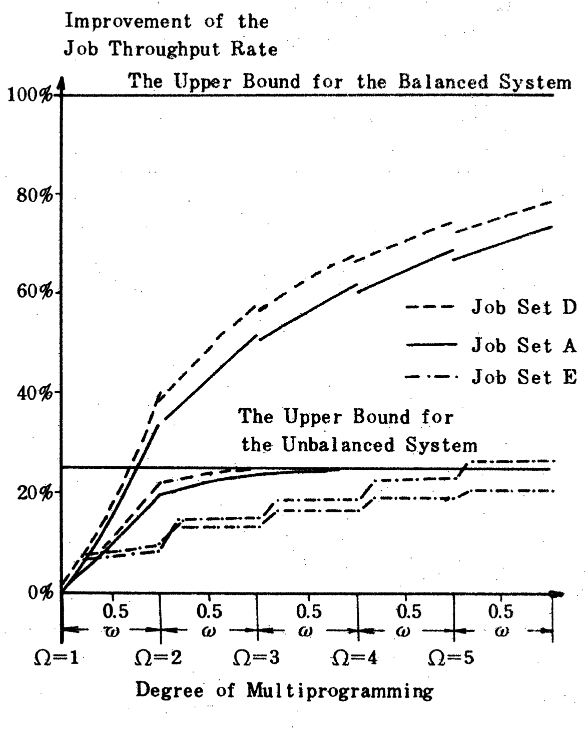


Figure 4.8 Improvement of the Job Throughput Rate under Nonpreemptive Schedules in a Balanced System and in an Unbalanced System

Finally, changes due to system balance are observed in Figure 4.8, job sets A, D and E are plotted for a balanced system where the CPU and the I/O channel are equally demanded, that is, λ/μ is equal to (1-q), and for an unbalanced system where I/O processor time is demanded four times more than CPU time, that is, λ/μ is equal to (1-q)/4. Queuing disciplines for the CPU are nonpreemptive, and each system has a single CPU and a single I/O channel. Observe that there is less improvement of the job throughput rate when the system is unbalanced. Also, the convergence of the job throughput rate as the degree of multiprogramming increases is more clearly observed when the system is unbalanced. When the distribution of CPU time is hyperexponential with a high variance, the improvement of the job throughput rate is small regardless of system balance at least for a small number of jobs in the system.

In conclusion the observations made in these examples are summarized as follows:

1. An increase in the job throughput rate can be obtained by increasing overlap within a job.
2. Under nonpreemptive schedules, having a high overlap ratio is about equivalent to the addition of another job in the system regardless of the distributions of CPU time and I/O time.
3. The job throughput rate is significantly increased by time-slicing. This is observed even if the distribution of CPU time is exponential.
4. When the distribution of CPU time is exponential, having a high overlap ratio under processor-sharing schedule is approximately equivalent to doubling the number of jobs in the system.
5. When the distribution of CPU time is hyperexponential with a high variance, small amount of overlap within a job has an equivalent effect to total overlap within a job. This is observed under preemptive schedules as well as nonpreemptive schedules.
6. The distribution of CPU time has significant effect on the job through rate of the system.
7. The distribution of I/O time appears to have less effect on the job throughput rate than the distribution of CPU time.
8. Queuing disciplines for the I/O processors appears to have little effect on the job throughput rate.
9. The job throughput rate per CPU is higher in a multiple CPU system than a single CPU system for the same degree of multiprogramming.
10. The less balanced the system is, the less improvement of the job throughput rate is obtained by multiprogramming and/or overlap within jobs.

References

[1] F. Baskett, III, Mathematical Models of Multiprogrammed Computer Systems, Ph. D. Dissertation, University of Texas at Austin, 1970.

[2] J. W. Boyse, Execution Characteristics of Programs on a Page-on-demand System, Comm. of ACM, 17, 4 (April 1974), pp. 192-196.

[3] D. L. Boyd, A Multiple Resource Model for a Batch-Processing Multiprogramming System, Tech. Report No. 39, Department of Mathematics, University of Iowa, March 1971.

[4] D. R. Cox and W. L. Smith, Queues, Methuen and Co., Ltd., London, 1961.

[5] W. Feller, An Introduction to Probability Theory and Its Applications, Vol. 1, 3rd Ed., John Wiley and Sons, Inc., New York 1968.

[6] D. P. Gaver, Probability Models for Multiprogramming Computer Systems, Journal of the ACM, Vol. 14, No. 3 (July 1967), pp. 422-438.

[7] P. A. Houle, Jr., A Study of Performance Driven Scheduling in a Multiprocessing Computer System, Ph. D. Thesis, University of Minnesota, 1973.

[8] L. Kleinrock, Time-Shared Systems: A Theoretical Treatment, Journal of the ACM, Vol. 14, No. 2, (April 1967), pp. 242-261.

[9] M. Maekawa, and D. L. Boyd, A Model of Concurrent Teaks Within Jobs of a Multiprogramming System, Proceedings of the Eighth Annual Princeton Conference on Information Sciences and Systems, Princeton University, Princeton, New Jersey, March 28-29, 1974, pp. 97-101.

[10] M. Maekawa and D. L. Boyd, Two Models of Task Overlap Within Jobs of Multiprocessing Multiprogramming Systems, Tech. Report 74-6, Department of Computer, Information and Control Sciences, University of Minnesota, March 1974.

OPTIMAL SCHEDULING OF VECTOR COMPUTATIONS IN A RECONFIGURABLE SHARED-RESOURCE ARRAY PROCESSING SYSTEM[a]

Alexander Thomasian and Algirdas Avizienis
Computer Science Department
University of California, Los Angeles 90024

Summary

Large, high-bandwidth main memories constitute a component of significant cost in current high-performance vector processors such as the STAR-100, the ASC and the CRAY-1. To relieve the main memory from instruction accesses, the streaming mode of operation is incorporated in all three computers. The CRAY-1 computer additionally provides register files in the CPU to relieve the memory from saving and refetching temporary results in vector operations. The handling of vector temporary results has turned out to be a major problem in the other two systems. The Shared Computing Resource (SCR) is another scheme to speed up vector operations by efficiently utilizing memory bandwidth [1].

The SCR system consists of an array of pipelined arithmetic processors (AP's). The AP's are interfaced with the main memory by means of address generators (AG's) which handle the fetching and storing of vector operands with respect to the main memory. The system is highly reconfigurable and a control unit sets up the required interconnections and registers for a given computation. The computation (task) which generally corresponds to the evaluation of a vector expression proceeds autonomously in streaming mode until termination. During the execution of a task, the AP's are interconnected so that they transmit intermediate vector results directly to each other.

Tasks to be run on the SCR originate from programs executing in a multiprogramming/multiprocessing system. The SCR serves as a shared-resource for vector processing in this system and several independent tasks originating from different programs can proceed concurrently in the SCR. Tasks correspond to the data-flow graphs of blocks of vector assignment statements which are executable by the SCR in vector mode. The nodes of the data-flow graph, which is a directed acyclic graph (dag), are either input or computational nodes. The edges of the data-flow graph correspond to the transmittal of input or temporary results. A memory access cost is associated with each edge. The cost is one memory access per vector element for edges emanating from input nodes. For temporary results the cost is two memory accesses per vector element, since they have to be stored and refetched.

Potentially a data-flow graph can be executed as a single task, by mapping it into a configuration of the SCR system (the AP's and the AG's). Since the resources are finite and since there is limited intercommunication among AP's, this mapping is not possible in all cases. Hence we allow each task to request at most p out of m AP's, where p is a design parameter. The completion rate of vector computations can be increased if the partitioning of the data-flow graph is performed such that memory accesses (partitioning cost) is minimized in the case when memory bandwidth is the limiting factor on speed.

We face the issue of partitioning the data-flow graph into subgraphs (tasks) such that each subgraph has at most p nodes and the cost of edges connecting the subgraphs is minimal. An efficient graph-partitioning algorithm exists when the dag corresponding to the data-flow graph is ordered linearly, the nodes are assigned consecutive numbers, and a task is restricted to consist of consecutively numbered nodes [2]. The linear ordering of the dag corresponds to the original order in which operations were specified in the input program. For examples of the application of this algorithm to the problem at hand the reader is referred to [3].

A tradeoff exists between the maximum allowable task size (p) and additional memory accesses made necessary by the need to save intermediate vector results. The tradeoff is studied by performing measurements on existing programs written for vector computers to ascertain the desirability of the SCR design and to optimize its parameters. To determine the maximum task size (the value of p), a static analysis of a set of benchmark programs is performed by partitioning the data-flow graphs of vector computations occurring in the programs. Then we determine the maximum task size, which while maintaining memory accesses at a low level, requires a moderate intercommunication scheme among the AP's.

[a]This research was supported by the National Science Foundation, Grant No. MCS72-03633 A04.

References

[1] A. Thomasian and A. Avizienis, "A Design Study of a Shared-Resource Computing System," Proceedings of the Third International Symposium on Computer Architecture, Clearwater, Florida, January 1976, pp. 105-112.

[2] B.W. Kernighan, "Optimal Sequential Partitions of Graphs," JACM, Vol. 18, No. 1, January 1971, pp. 34-40.

[3] A. Thomasian, A Design Study of a Shared-Resource Array Processing System, Ph.D. Dissertation, Computer Science Department, UCLA, September 1976.

A MODEL FOR A SHARED RESOURCE MULTIPROCESSOR

by

Lawrence S. Cheung
Department of Electrical Engineering
Marquette University
Milwaukee, Wisconsin 53233

and

Frederic J. Mowle
Department of Electrical Engineering
Purdue University
W. Lafayette, Indiana 47907

Abstract -- A model for a shared resource multiprocessor is presented. Based on this model, two different problems are investigated. The first problem is to study the system utilization and the response time as a function of the number of programs sharing a given set of resources. The second problem is to develop an algorithm for determining the minimum (cost) system configuration for a given environment.

Throughout this paper, the significance and importance of system balance and its relation to resource utilization are emphasized.

The Need For A Shared Resource Multiprocessor

The increasing demand for computers with large computing power and high reliability has led to the concept of modularity. The availability of large scale integrated circuits also makes this approach more attractive.

In designing a processor, system architects [3] have explored and investigated the idea of dividing a processor into two separate units, an instruction fetch (I) unit and an execution (E) unit. The I unit is responsible for fetching instructions from the memory and sending them to the E unit for execution. The E unit performs all the arithmetic and logic operations as instructed by the I unit. In some cases, the E unit is further subdivided into several independent functional units (e.g. IBM 360/91 and CDC 7600) with each functional unit responsible for the execution of a special group of instructions. This kind of specialization may increase the computing speed, but it may also lead to the problem of low hardware utilization. At any moment, only a small percentage of all the available resources is not idle.

In order to increase the hardware utilization to an acceptable level, Flynn has proposed the idea of shared resources [2]. Suppose a processor has a total of X functional units. At some instant, task A only requires i units. If there is another task B, which can use some of the X - i remaining units, then the overall hardware utilization can be improved. In another word, we are asking the X units to serve more than one program or task in order to increase the demand for the hardware resources.

It may happen at times that both task A and task B are requesting the same resource, in which case a priority scheme will be needed to resolve the conflict. Due to the possibility of resource contention, the time needed to execute task A may be longer in a shared resource environment, however the time needed to execute both task A and task B will be shorter than if they had been run sequentially. This is an example of a trade-off between response time and system throughput. If the system is properly designed, the sacrifice in response time to obtain high system throughput can be kept at a minimum.

Program scheduling can also play an important part in increasing system throughput. If task A and task B have two different resource characteristics, e.g. task A requires a lot of floating point arithmetic while task B only works on non-numeric data, then the resource conflict can be kept at a very low level.

Balance Of A Highly Parallel System

Instead of looking at a highly parallel processor as a web of specialized units, it can be analyzed based on the functions or operations[a] it performs, e.g. instruction fetch, add etc. Each operation requires a set of inputs and produces an output. The output of an operation may also be the input to another operation. Some of these functions have to be performed frequently, while others will be needed only occasionally.

[a] Here we assume a processor to be a collection of functional units interconnecting together in some manner.

In designing a highly parallel processor, after identifying all the operations ($F_1, F_2, \ldots, F_n$) it has to perform, the next step is to partition them into L distinct groups ($G_1, G_2, \ldots, G_L$) such that each group of operations can be carried out by a class of specialized functional units. Each class i contains N_i independent functional units. A class i functional unit is only capable of performing the operations in G_i. Associated with each operation F_j, we define T_j to be the time needed to perform F_j. Using this notation, a processor can be described by (S_L, $N_1, \ldots, N_L$, $T_1, \ldots, T_n$) where S_L is the partition of $F_1, \ldots, F_n$, into L groups, $G_1, G_2, \ldots, G_L$.

After defining $F_1, \ldots, F_n$, a program can be characterized by w_i, $i = 1,2,\ldots,n$, the probability that an instruction in the program requires F_i. For a given partition S_L, the probability that an instruction requests the service of a functional unit in class i is

$$P_{iS} = 1 - \prod_{F_j \in G_i} (1 - W_j).$$

The demand by the program on the class i functional unit is

$$X_{iS} = P_{iS} \; T_i / N_i .$$

and the fraction of the total demand on the class i functional unit is

$$W_{iS} = X_{iS} / (X_{1S} + X_{2S} + \ldots + X_{LS}).$$

Given a processor with a partition S_L, we can define D_S, the degree of balance, as

$$D_S = \sum_{i=1}^{L} (W_{iS} - 1/L)^2.$$

This is a measure of how even the load on the system is distributed to various classes of functional units. S_L will be called perfectly balanced if $D_S = 0$. It should be noted that a perfectly balanced partition may not exist. A partition S_L is called balanced if for any other partition V_L of a design, D_V is greater than or equal to D_S. A processor is (perfectly) balanced if its operations are grouped together according to a (perfectly) balanced partition.

In a shared resource multiprocessor environment, the L classes of functional units will be shared among all the active programs. If there are M active programs with the same characteristics, the contention factor, C_j, is defined to be

$$C_j = \sum_{i=2}^{M} \binom{M}{i} W_{jS}^{i} (1 - W_{jS})^{M-i}.$$

This is a measure of the contention for the class j functional units by the M programs. The total contention factor, TC_S, for the partition S_L is

$$TC_S = \sum_{i=1}^{L} C_i .$$

In a shared resource environment, the resource contention by different programs should be minimized as much as possible; in other words, we want to find a partition S_L of F_i's such that TC_S is minimized. Assuming the T_i's can be defined in such a manner that W_{jS} can take any value between 0 and 1, then the question can be formalized as an optimization problem.

Minimize

$$TC_S = \sum_{j=1}^{L} \sum_{i=2}^{M} \binom{M}{i} W_{jS}^{i} (1 - W_{jS})^{M-i}$$

subject to the constraints

$$\sum_{j=1}^{L} W_{jS} = 1$$

$$1 > W_{jS} > 0 \qquad \text{for } j = 1,2,\ldots,L.$$

This can be solved by using the Lagrange multiplier and the solution is

$$W_{jS} = 1/L \qquad \text{for } j = 1,2,\ldots,L.$$

i.e., a perfectly balanced partition will result in a minimum TC_S.

However, in most cases, a perfectly balanced partition does not exist. Therefore, a realistic goal is to find a partition S_L which is "almost" perfectly balanced. D_S is a measure of the deviation of S_L from the ideal case, and therefore, it is the balanced partition that we are looking for. This can be found using perturbation or exhaustive search.

Note that the execution times for various operations are dependent on the design of the specialized units. For example, a "bit by bit" shifter may take several times longer to execute an instruction than a variable length shifter. Therefore, the structures of the specialized units are also a very important factor in the design of a parallel system. If some functions are used infrequently, it is not cost effective to implement such functions using specialized units. It is best to invest into the area where the return is the greatest.

A Model For A Shared Resource Multiprocessor

In discussing the balance of a processor, a very crude model was used. No consideration was given to the system throughput and the response time. They are the basic concerns in this section. The model presented in the sequel can be used to study how the overall system reacts as the number of programs sharing a given set of resources increases.

The structure assumed consists of M I units sharing a single E unit. The I units are responsible for all the control and sequencing necessary to execute the instructions of the programs and the E unit is responsible for carrying out all the arithmetic and logical operations. Every cycle, each I unit looks at the first K instructions from a program or an instruction stream, determines all the independent instructions, decodes them, and sends the appropriate signals and data to the E unit. When there are not enough functional units to serve all the instructions they will be stored in buffers until the appropriate functional unit is free.

The following assumptions are made on the system configuration:

1) All the I units are synchronized. They all decode instructions and request services from the E unit at the same time.
2) Instructions from different I units are assumed to be independent of each other.
3) The E unit consists of L distinct classes of functional units, such as multiply unit, Boolean unit, and divide unit. Class i has N_i identical functional units. Each functional unit is capable of independently carrying out a specific class of instructions.
4) No instruction requires more than one service from the E unit.
5) The execution time for all the instructions in the same class is the same.
6) The cycle time for a functional unit is an integral multiple of some unit time.
7) For each class of functional units, there is a buffer of infinite length to hold all the service requests. (This assumption is valid if the size of the buffer is at least as great as the number of functional units in class i).

8) The probability, Q_{ij}, of sending i independent instructions from an I unit to the E unit when there are j instrucitons from the same I unit still active in the E unit is given by

$$Q_{ij} = ab^{i+j-1} \quad i = 1,2,\ldots,K$$

$$Q_{0j} = 1 - \sum_{i=1}^{K} Q_{ij}$$

where a and b are some constants less than one (See [1] for a justification of this assumption) and K is the number of instructions examined by an I unit in a cycle.

9) After the arrival of an instruction, it takes one time unit for it to stabilize in the buffer before it can be processed by a functional unit. Therefore, the minimum time an instruction stays in the system is its execution time plus one.
10) Each instruction from the I unit has a probability, p_i, of requesting the service of a class i functional unit.
11) All instructions sent to the E unit are assumed to have the same priority, regardless of their origin.
12) All instruction streams are assumed to have the same characteristics, i.e. Q_{ij} and p_i are assumed to be the same for all programs.

Based on these assumptions, a shared resource multiprocessor can be studied using a queueing model (Fig. 1) with multiple sources (I units), multiple infinite queues (one queue or buffer for each class of functional units) and multiple servers (functional units). Each service station may have more than one stage and the number of stages is equal to the number of time units needed to carry out the service. Since pipelining within each functional unit is not assumed, each service station can only accomodate one customer. Normally there is no waiting room between stages.

Notation

L	Number of classes of functional units
N_i	Number of functional units in class i
T_i	Cycle time for class i functional units
M	Number of I units in the system
K	Maximum number of instructions decoded per cycle
Q_{ij}	The probability of sending i instructions to the E unit from an I unit when there are j instructions from the same I unit still active in the E unit
p_i	The probability of an independent instruction sent by an I unit requests the service of a class i functional unit

Analysis of the Model

This model can be solved mathematically; but it is not appropriate here because the model is too complicated and a set of complex algebraic equations does not offer any insight. Furthermore, there does not exist any closed form solution to the generalized model which is the chief advantage of a mathematical analysis. Therefore, simulation was chosen to study the model.

However, a simple solution does exist in the limiting case when the number of I units, M, approaches infinity. Let's define the demand by an environment on the class i functional units to be

$$X_i = p_i T_i / N_i \qquad i = 1,2,\ldots,L.$$

If

$$Y = \mathrm{Max}(X_1, X_2, \ldots, X_L),$$

we can define the normalized demand on the class i functional units to be

$$Z_i = X_i/Y.$$

As M approaches infinity, the class of functional units with the largest X becomes the bottleneck in the system and its utilization factor approaches 1. Under this circumstances, Z_i is the normalized work done by the class i functional units and, therefore, is also the utilization factor for the class i functional units. The limiting system utilization factor (LUF) as N approaches infinity is

$$LUF = \frac{\sum_{i=1}^{L} Z_i N_i}{\sum_{i=1}^{L} N_i} . \qquad (1)$$

Referring back to the section on system balance, the goal there was to minimize contention among programs so as to decrease system response time and increase system utilization. These factors are interrelated and an alternative goal, to maximize LUF, can also be used. Note that Z_i's are related to the W_{iS}'s defined before. Both of them are measures of the demand for the class i resource. Therefore, it is not surprising that LUF is maximized when $X_1 = X_2 = \ldots = X_L$.

The example used here to illustrate the above principles consists of two classes of functional units. The cycle time for the first and second class of functional units are 3 and 1 time units respectively. The example can be described by the following parameters:

$$L = 2 \qquad K = 4$$
$$T_1 = 3 \qquad T_2 = 1$$
$$P_1 = 0.1 \qquad P_2 = 0.9$$
$$Q_{ij} = 0.4(1/2)^{i+j-1} \quad \text{for } i = 1,2,\ldots,K$$
$$Q_{0j} = 1 - \sum_{i=1}^{K} Q_{ij}.$$

Three different cases are considered. In the first case, there are one class 1 functional unit ($N_1 = 1$) and three class 2 functional units ($N_2 = 3$). The limiting system utilization factor is

$$LUF_1 = \frac{1 + 1 \times 3}{4} = 1.0.$$

In the second case, there are two functional units in each class ($N_1 = N_2 = 2$) and the limiting system utilization factor is

$$LUF = \frac{(.15/.45) \times 2 + 1 \times 2}{4} = 2/3$$

In the third case, there are three class 1 functional units ($N_1 = 3$) and only one class 2 functional unit ($N_2 = 1$). The limiting system utilization factor is

$$LUF_3 = \frac{(.1/.9) \times 3 + 1}{4} = 1/3.$$

For all these cases, the number of functional units is a constant, but the LUF ranges from 1 to 1/3 (from a perfectly balanced system to a highly unbalanced system). The idea is to see how the system reacts under different loading conditions. For each case, five simulation runs were performed with different numbers of I units (M = 1, 2, 4, 8, 12). The results are shown in Fig. 2 and Fig. 3. Two different measures are used, the utilization factor and the response time. The values shown in Fig. 3 are normalized against the situation when there is only one I unit (M=1).

The results obtained are somewhat expected. The utilization factor increases "almost linearly" for all cases until the system becomes saturated and is bounded by the LUF as predicted. Response time also increases as M increases. The performance of case (1) is better than case (2), and case (2) is better than case (3). This again emphasizes the importance of the balance of a system.

Regarding the trade-off between response time and system utilization, it is hard to define an optimal point because they are two different things. However, some general guideline can be obtained from the model. For example, in case (1), the utilization factor increases from 0.2545 to 0.8242 as M increases from 1 to 4 while the response time only increases by 12.3%. This is a price many people are willing to pay.

The analysis shown is centered on the processor only and has ignored all the other components of a computer, such as memory and peripheral devices. This model can be extended to include peripheral devices by treating them as a special kind of functional units. In this case, Q_{ij} may have to be redefined to take into account the fact that an I unit may be idle while waiting for I/O.

When the response time and M are increased, a program has to stay in the memory longer and the memory size has to increase in order to accomodate more programs. Since memory is one of the most important resources in a computer, this should be accounted for when considering resource utilization. As it is directly related to response time, this fact can be taken care of by assigning appropriate weighting factor to system response time during the design.

In the analysis we have explicitly assumed that the costs for all kinds of functional units are equal. In practice, this is not true. It may be more suitable to use system utilization per unit cost as a performance criterion. If so, X_i and LUF can be redefined to be

$$X_i = p_i T_i C_i / N_i ,$$

$$LUF = \frac{\sum_{i=1}^{L} Z_i N_i}{\sum_{i=1}^{L} N_i C_i} , \qquad (2)$$

where C_i is the cost of a class i functional unit.

A Design Problem

The model discussed can also be used to aid the design of a shared resource multiprocessor system. Consider the optimization problem, given M, L, T_i, K, C_i, Q_{ij} and p_i, minimize

$$CF = \sum_{i=1}^{L} N_i \qquad (3)$$

subject to the constraint

instruction execution rate $\geq \alpha$.

where α is a constant.

Instruction execution rate is used as a measure here because it gives a more precise description of the capability of the system. Response time and utilization factor, though directly related to instruction execution rate, are only good as relative measures, but too vague to be an absolute measure.

Since the model is based on simulation, in order to obtain the optimal solution, the only way is to use an enumerative approach. However, certain criteria can be formulated to reduce the number of trials.

Since $1/T_i$ equals the number of instructions a class i functional unit can execute in a cycle and the total capacity of all the functional units must be greater than or equal to α, this can be expressed as

$$\sum_{i=1}^{L} \frac{N_i}{T_i} \geq \alpha .$$

Since the load is distributed among different classes according to the probabilities p_i, more precisely, we can write

$$\frac{N_i}{T_i} \geq p_i \alpha \qquad i=1,2,\ldots,L . \qquad (4)$$

The second criterion is based on the conjecture that a balanced system always outperforms a unbalanced system. Therefore, whenever it is necessary to add one additional functional unit to the system, add to the class such that the resulting LUF is a maximum.

The algorithm in determining the optimal solution is outlined as follows:

(1) Let $N = (N_1, N_2, \ldots, N_L)$ such that N_i is the smallest integer that eq. (4) is satisfied.

(2) Carry out the simulation to find out whether the constraint is satisfied or not. If not, go to step 3; otherwise stop.

(3) Find N_i such that the corresponding X_i is a maximum, i.e. $X_i = \text{Max}(X_1, X_2, \ldots X_L)$ and increment N_i by one. (This is equivalent to add one additional functional unit to class i with the resulting LUF a maximum). Go to step (2).

If we use the parameters in the previous example and let $\alpha = 2$, the possible sequence of combinations one may try is (1,2), (1,3), (1,4), (2,4), (2,5), (2,6) ---. Since the constraint is satisfied for N = (1,3), in actual carrying out the algorithm, only two trials have to be performed.

If one is interested in the cost of the system rather than the total number of functional units used, eq. (3) can be replaced by

$$CF = \sum_{i=1}^{L} N_i C_i$$

and LUF should be defined as in eq. (2) instead of eq. (1).

Since the sequence of the possible combinations is known beforehand, one can make an intelligent guess and use that as a starting point. In many cases, this can cut down the number of trials and speed up the algorithm.

Discussion and Conclusion

Throughout this paper, we have assumed that all the programs have the same characteristics (assumption 12). If this assumption is relaxed, eq. (1) is no longer correct since different programs may impose different loads on the system. This case will be considered in a future paper.

Another observation is the utilization factor for a shared resource multiprocessor increases almost linearly for small M. For a given set of functional units and M, one can estimate roughly the utilization factor by determining the LUF and the utilization factor when M = 1.

The model presented in this paper is useful in analyzing and designing a shared resource multiprocessor system. The notion of system balance and the importance of LUF are discussed and emphasized. A design problem is also formulated and an algorithm for solving this problem was proposed.

Bibliography

(1) Cheung, L. S., "Techniques for Reducing Dependencies among Instructions for a Parallel Single Processor Computer System," Ph.D. thesis, Department of Electrical Engineering, Purdue University, 1975.

(2) Flynn, M. J., "Some Computer Organization and Their Effectiveness", IEEE Trans. Computer, C-19, 10 (Oct. 1970), 889-895.

(3) Lorin, Harold, "Parallelism in Hardware and Software", Prentice-Hall Inc., Englewood Cliffs, N.J., 1972.

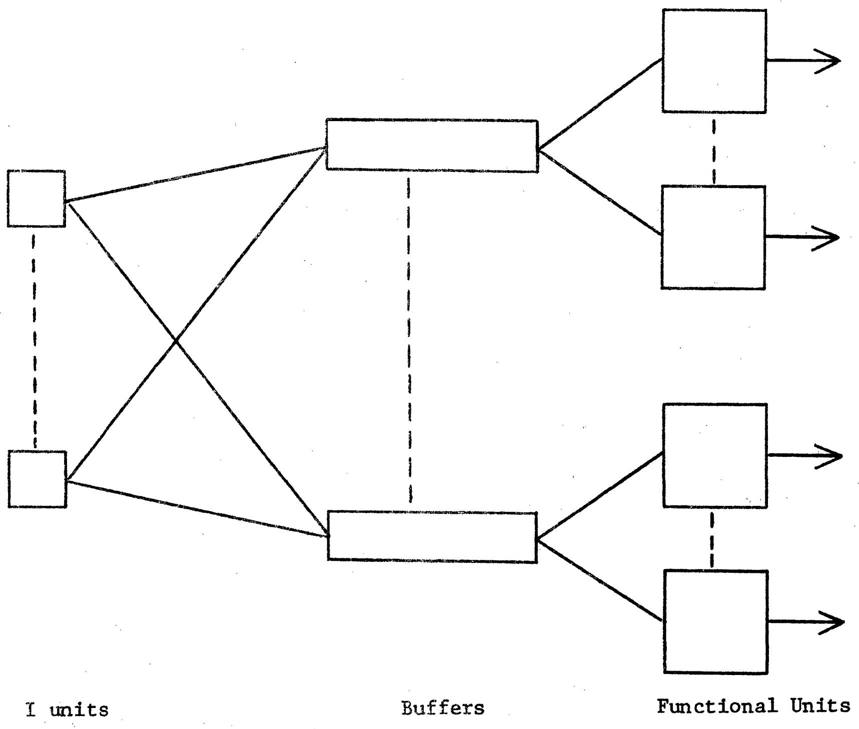


Fig. 1. A queueing model for a shared resource multiprocessor.

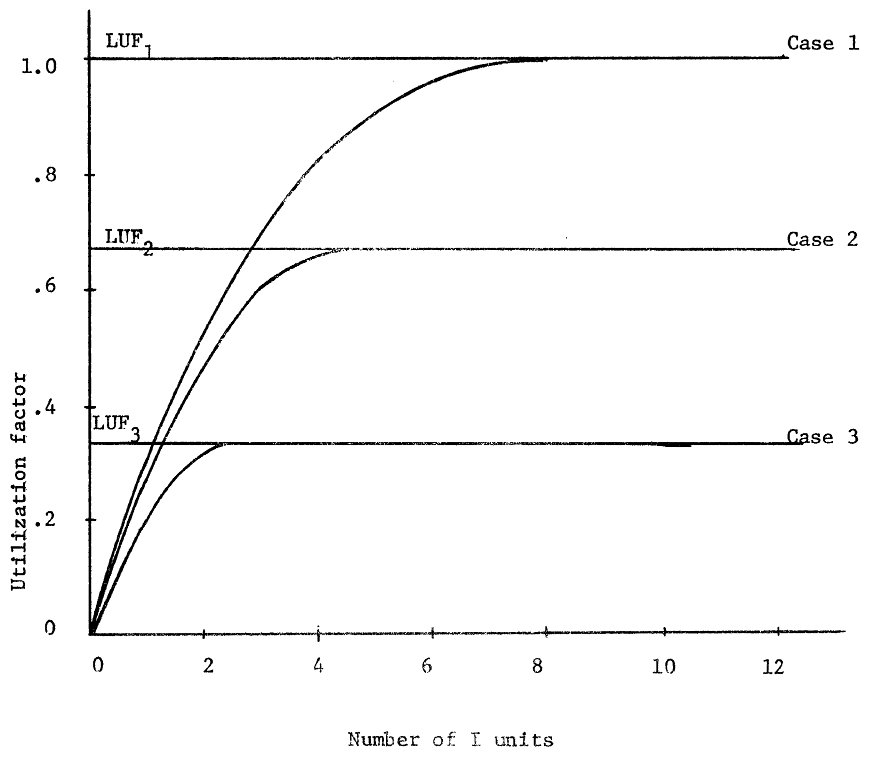


Fig. 2. Ultization factor as a function of the number of I units.

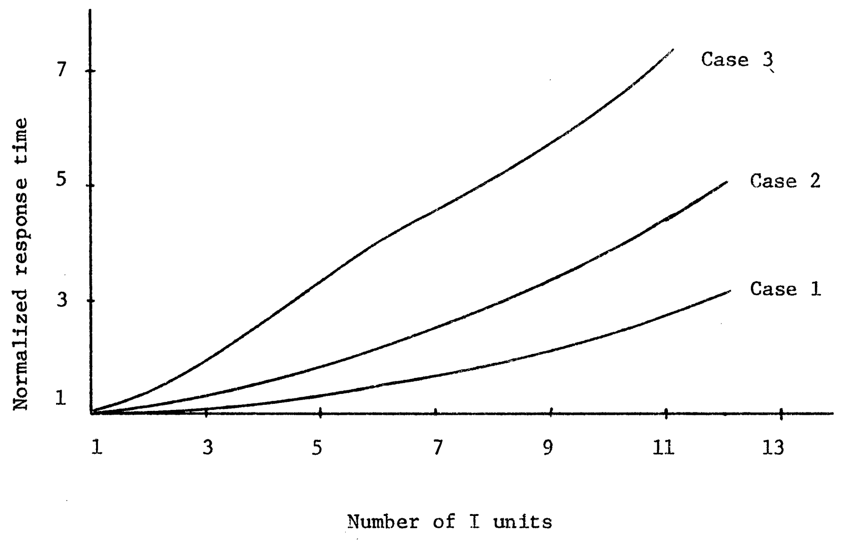


Fig. 3. Normalized response time as a function of the number of I units.

PERFORMANCE ANALYSIS OF A DATA-FLOW PROCESSOR*

David P. Misunas
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- A data-flow processor is structured as a packet communication system. Sections of a processor are connected by interconnection networks which have a great deal of inherent parallelism, and the sections communicate by means of fixed size information packets. The processing capability of a data-flow processor is determined through consideration of the flow of packets within the interconnection networks, and the actual performance of the processor is affected by the structure of the networks. The execution time of an instruction in a processor can vary greatly due to conflict within the interconnection networks. The performance of a data-flow processor is measured through consideration of the delays caused by this conflict, and the proper network structure and processing rate of a machine are determined through analysis of the best and worst case delays.

Introduction

Efforts to develop a model of computation which can effectively express parallelism have yielded a new form of program representation known as data flow [1,2,3,6,7,8,10]. The attractiveness of data flow lies in the fact that it is data-driven; that is, an instruction is enabled for execution only after each required operand has been provided by the execution of a predecessor instruction.

We have been conducting architectural studies to investigate the design of a processor which can efficiently execute data-flow programs by taking advantage of the parallelism inherent in the data-flow representation. The resulting architectures [4, 5] offer attractive solutions to some of the problems of parallel systems. The usual problems of processor switching and memory/processor interconnection are avoided by the use of interconnection networks which have a great deal of inherent parallelism. The structure of the processor allows a large number of instructions to be active simultaneously. These active instructions pass through the networks concurrently and form streams of instructions for the pipelined functional units.

Initial investigations culminated in the development of an architecture for a processor that executed programs expressed in the elementary data-flow language [4]. The elementary language incorporates no fancy capabilities such as recursion, data structures, conditionals, or iteration. However, the language and its corresponding architecture are well-suited for the representation and execution of signal processing computations such as filtering, waveform geveration, fast Fourier transforms, and so forth.

The next step involved developing the architecture of the basic processor [5]. This machine and its corresponding language incorporate conditional and iterative mechanisms and a multi-level memory system in which the active memory is operated as a cache, and individual instructions ar retrieved from the auxiliary memory as they become required for computation.

The most recently developed machine in this series expands the architecture and language to incorporate procedures, recursive activation, and data structures represented as acyclic directed graphs [8, 9]. A more conventional approach to the implementation of a complete data-flow language has been developed by Rumbaugh [11, 12].

The performance of a data-flow processor is analyzed through consideration of the flow of information within the interconnection networks of the processor. In illustration of this technique of performance analysis, we consider such an analysis of the performance of an elementary data-flow processor.

The Elementary Data-Flow Processor

The computational capability of the elementary data-flow processor is limited to programs expressed in the elementary data-flow language. A program in this language is constructed of two kinds of elements, called operators and links. Operators are represented as circles with a number of input arcs and one output arc. A link is designated by a small dot and receives results from an operator on its input arc and distributes them to other operators over its output arcs.

Tokens are represented by large solid dots and convey values over the arcs of the program. An

*This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-06661.

operator with a token on each of its input arcs and no token on its output arc is enabled and sometime later will fire, removing the tokens from its input arcs, computing a result using the values associated with the input tokens, and associating that result with a token placed on its output arc. Similarly, a link is enabled when a token is present on its input arc and no token is present on any of its output arcs. It fires by removing the token from its input arc and associating copies of the value carried by the input token with tokens placed on its output arcs.

In Figure 1 we have a rather simple data-flow program. There is a value present on each input arc, and thus links L1 and L2 are enabled. Either one can fire -- suppose L1 does. Then operator A2, which multiplies its input by the constant A, and link L2 are enabled. Once again, either A2 or L2 can fire, and in this manner tokens travel through the program until a token appears on the output conveying the value Ax(x+y). Once operators A1 and A2 have fired, there are no tokens on the arcs emanating from L1 and L2, and the links can fire as soon as two new input values arrive. Thus, these elementary programs can readily represent pipelined computation.

The Memory of the elementary data-flow processor shown in Figure 2 holds a representation of the program to be executed. This Memory is a collection of Instruction Cells (Figure 3); one Instruction Cell is associated with each operator of the program. Each Instruction Cell is composed of three registers, the first of which specifies the operation to be performed and the address(es) of the register(s) to which the result of the operation is

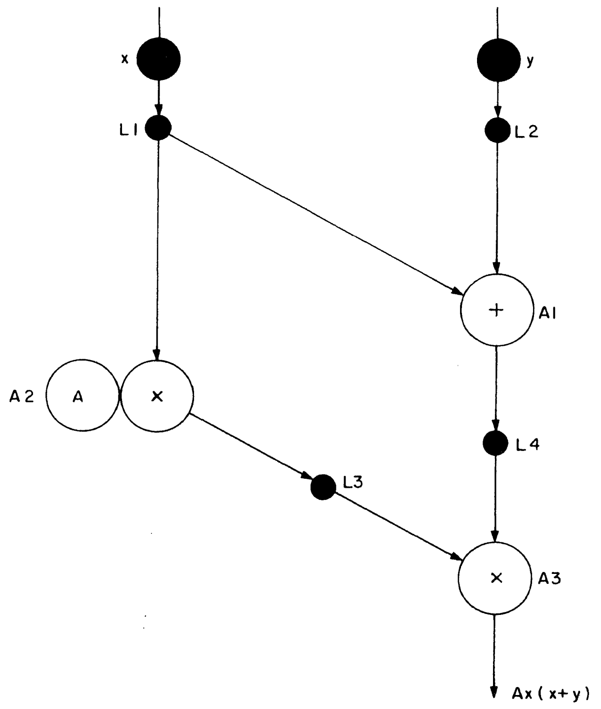


Figure 1. An elementary data-flow program.

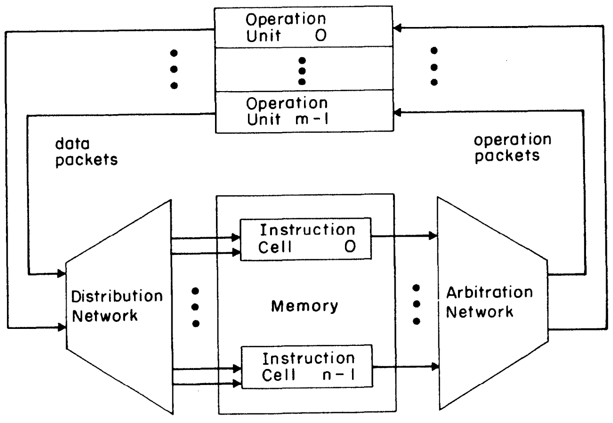


Figure 2. Structure of the elementary data-flow processor.

to be directed. The second and third registers receive operands for use in execution of the instruction.

When an Instruction Cell contains an instruction and all required operands, the Cell is said to be enabled and presents its contents as an operation packet to the Arbitration Network for delivery to an Operation Unit which can perform the desired function. The Arbitration Network provides a path from each Instruction Cell to each Operation Unit. The network is capable of simultaneously accepting many operation packets from the Instruction Cells and delivers each packet to an appropriate Operation Unit by decoding the instruction portion of the packet.

Upon receiving an operation packet, an Operation Unit performs the function specified by the instruction on the operands of the packet and produces a data packet, containing one copy of the result and a destination register address, for each destination specified in the instruction. A Distribution Network concurrently accepts data packets from the Operation Units and, using the destination address of each packet, delivers it to the specified register of the Memory. The

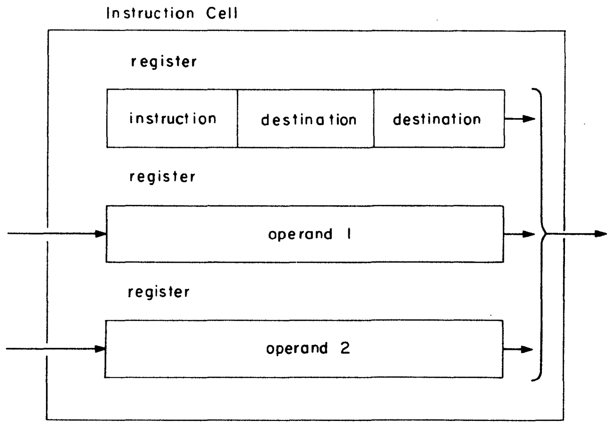


Figure 3. Structure of an Instruction Cell.

Instruction Cell containing that register may then be enabled if an instruction and all operands are present in the Cell.

A simplified structure of the Arbitration and Distribution Networks is presented in Figure 4. The networks are composed of three types of units. An arbitration unit passes packets arriving at its input ports one-at-a-time to its output port, using a round-robin discipline to resolve any conflicts. A switch unit passes a packet at its input to one of its outputs, controlled by some property of the packet. In the Arbitration Network this property is the operation code, whereas in the Distribution Network, the switch units are controlled by the destination address. A buffer unit stores a packet until the succeeding switch or arbitration unit is ready to accept it.

Due to the large number of inputs to the Arbitration Network, we wish to transfer data between the Memory Cells and the Arbitration Network in serial format to reduce the number of wires necessary. However, in order to maintain a high rate of packet flow at the output ports, we wish to transfer packets to the Operation Units in parallel format. For this reason, serial-to-parallel conversion is done gradually within the buffer units as a packet travels through the Arbitration Network. Parallel-to-serial conversion is performed in the Distribution Network for similar reasons.

Processor Performance

To analyze the performance of the elementary data-flow architecture, we must consider the utilization of the Instruction Cells of the Memory; that is, the number of times a Cell will be enabled within a given time period. This will then allow us to determine the processing rate of the machine.

The execution cycle time of an instruction within the processor is the minimum elapsed time between the enabling of the instruction and the arrival of the result of the operation specified by the instruction at the desired destination Cell(s For an instruction of the elementary data-flow processor, the execution cycle time is equal to the passage time through the Arbitration Network, the Distribution Network, and an appropriate Operation Unit. The delay in the Operation Unit is fixed for that Operation Unit. However, the network delays can vary greatly due to the presence of conflict.

The execution cycle time for an instruction is found by considering the passage of the operation packet containing that instruction through the Arbitration Network and the passage of the resulting data packets through the Distribution Network with no conflict. The minimum delay through a network, the Arbitration Network for example, is given by the summation over the number of stages in the network of the time required to transfer a packet through each stage:

(no. bits serial + 1)(bit transfer time)

The transfer time for a stage is equal to the number of bits passing through the stage in serial plus one for a signal to indicate that the packet is ready to be transferred multiplied by the time necessary to transfer a bit. A similar equation applies to delay in the Distribution Network.

Let us examine the delay within a specific Arbitration Network (Figure 5). This network has three stages and seven arbitration units. Packets travel through stage 0 in four-bit serial format and are gradually converted to a more parallel format, passing through stage 1 in two-bit serial and stage 2 in one-bit serial format. As noted previously, the passage time for a packet through each stage is equal to the number of serial bits plus one times the bit transfer time t. For the structure of Figure 5, the transfer times are 5t, 3t, and 2t, respectively. The minimum delay through the network is equal to the summation of the stage delays, or 10t.

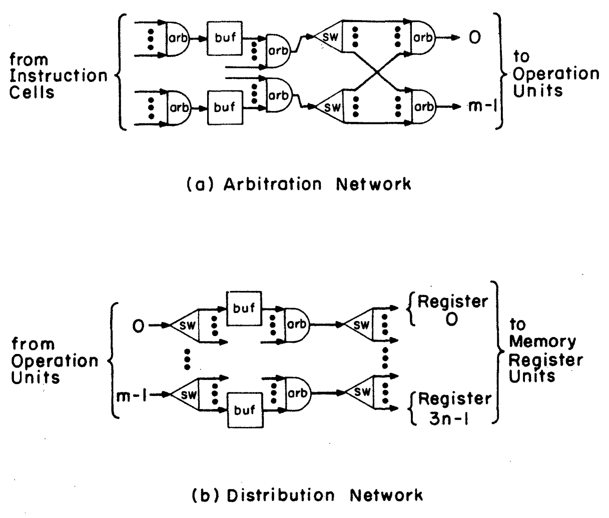


Figure 4. Structure of the Arbitration and Distribution Networks.

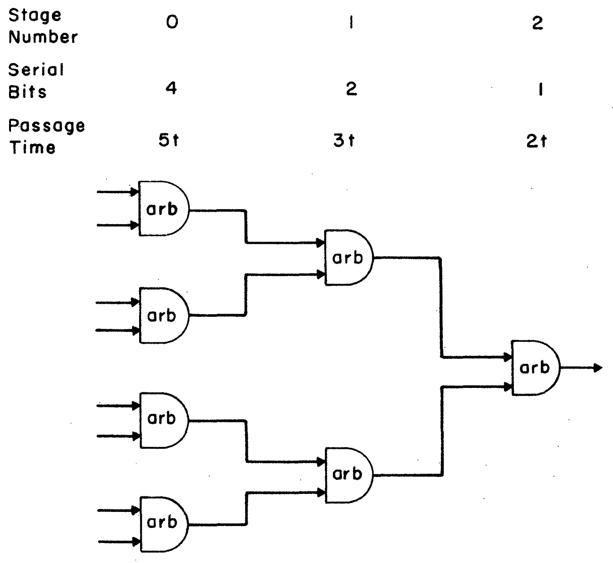


Figure 5. Structure of an elementary Arbitration Network.

To find the time T necessary to process all instructions contained in the Memory of the processor, we must consider the maximum delay a packet can encounter in passing through the Arbitration Network. Such a maximum delay can occur in a network which has a packet present at every node in a machine in which every Instruction Cell is enabled, placing a packet on each input to the Arbitration Network (Figure 6). The maximum delay which can be encountered by a packet, say the triangular one, arises only when all other packets in the network pass through the output of the network before the triangular one does. In order for this to happen, not only must the triangular packet lose every conflict, but every packet on the path it will follow to the output must also lose every conflict. Thus, finding the maximum delay involves determining how many packets will flow through each stage before the triangular one.

For this network, the worst case packet will be the 14th through stage 2, the 6th through stage 1, and the 2nd through stage 0. Multiplying the number of packets passing through each stage by the delay in that stage, we find that:

$$\begin{aligned} T &= \text{maximum delay} \\ &= 2(5t) + 6(3t) + 14(2t) \\ &= 56t \end{aligned}$$

Hence, if all instructions of the processor are enabled, they can pass through this Arbitration Network in a maximum time of 56t.

However, if we assume that the network size is such that the execution cycle time is less than T, then a number of destination Cells become enabled and enter the Arbitration Network before all Cells have been processed, and the processing rate of the machine can be measured in terms of the output rate of the Arbitration Network (assuming the Distribution Network has been structured to distribute all results as fast as they are produced). In such a case, the rate of packet transfer to each Operation Unit is 1/(2t), and the maximum processing rate of the machine is [1/(2t)](number of Operation Units).

Furthermore, if each arbitration unit has enough inputs to allow a packet to travel through the previous stage in less time than that required to service all busy inputs, the passage of the triangular packet through the first stages of the Arbitration Network will occur simultaneously with the transmission of other packets at the output of the network. The time T for the transmission of all packets in the network to the Operation Units is then 14(2t) = 28t.

Network Structure

The results developed in the previous section seem to indicate that a network of as few stages as possible is desirable in order to decrease the execution cycle time and increase the number of inputs to an arbitration unit of the network. In general, this is true. However, the fact that packets are transferred from each Instruction Cell in serial format requires a number of stages in the Arbitration Network in order to perform the conversion to parallel format before a packet

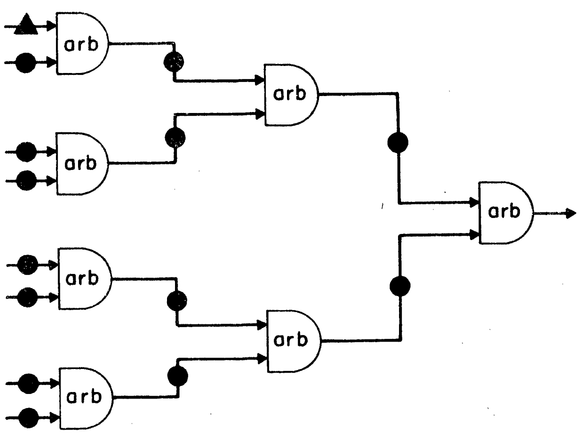


Figure 6. Example of a full Arbitration Network.

reaches the final stage of arbitration. Also, a number of stages are necessary in order to maintain a queue of instructions for each Operation Unit.

The actual structure of the Arbitration Network does not significantly affect performance as long as a few simple rules are observed in its construction. If D_{Ai} is the passage delay of a packet through stage i of the Arbitration Network, and I_{Ai} is the number of inputs to stage i, then the following relationship must hold:

$$D_{Ai} = \alpha[(I_{A(i+1)})(D_{A(i+1)})], \ \alpha<1$$

This assures that each stage of the Arbitration Network is kept busy by the preceding stages.

The value of the constant α is dependent upon the utilization of the machine. Since the processor is designed to support pipelined computation, the value of α is controlled by the amount of the machine which is used for computation and the difference between the sample input rate and the maximum processing rate.

The addition of a switch unit at the output of an arbitration unit introduces a further factor for consideration. If S_{Ai} is the number of outputs of the switch unit after stage i of arbitration, then

$$D_{Ai} = \alpha[(I_{A(i+1)})(D_{A(i+1)})]/S_{Ai}$$

and the number of inputs to the arbitration units of stage i+1 must be increased by the number of outputs of the switch unit of stage i in order to keep the arbitration unit in stage i+1 busy.

Similarly, the Distribution Network must be structured so that

$$D_i = \alpha[(S_i)(D_{(i+1)})]/I_i$$

where S_i is the number of outputs of the switch unit in stage i, I_i is the number of inputs of the arbitration unit preceding the switch unit of stage i, and D_i is the delay through stage i of the network.

An Example Processor

In illustration of the capability of an elementary data-flow processor, consider the execution of a highly parallel, pipelined computation on a 128 Instruction Cell machine in which all Cells are fully utilized. The Instruction Cells of the example machine accept and transmit packets in 16-bit parallel, 4-bit serial format.

For a balanced processor structure, one in which the number of Operation Units is matched to the number of Instruction Cells, the processing time T should be equal to the minimum delay D through the networks and an Operation Unit. Thus, to determine the optimal number of Operation Units for the processor, we must consider the structure of the networks in order to discover the minimum delay.

To obtain a small execution cycle time, and hence, a greater processing capability, the networks must be structured with as few stages as possible. However, three stages are required in the Arbitration Network to perform the serial-to-parallel conversion and still maintain the necessary throughput from stage to stage. The minimum delay analysis of this three stage network structure is identical to that described in the previous section; the delay in the Arbitration Network is equal to 10t.

Assuming that the minimum delay in the Distribution Network and the delay in an Operation Unit are the same as that in the Arbitration Network, the resulting value for D is:

$$D = 30t$$

If t = 150 nanoseconds, allowing 15 TTL gate delays to accomplish one ready/acknowledge cycle, the resulting execution cycle time is :

$$\begin{aligned} D &= 30(150 \text{ nsec.}) \\ &= 4.5 \text{ microseconds} \end{aligned}$$

To establish the number of Operation Units necessary for a balanced processor structure, with a stage delay of 300 nsec. for each pipelined Operation Unit, we must set the processing time T for all enabled instructions contained in the Memory equal to the execution cycle time:

$$\begin{aligned} T &= 4.5 \text{ microseconds} \\ &= (128)(300 \text{ nsec.})/(\text{no. of Operation Units}) \end{aligned}$$

yielding:

$$\text{no. of Operation Units} = 9$$

And the resulting performance of the processor is:

$$\begin{aligned} \text{processing rate} &= 128 \text{ instructions } / \text{ } 4.5 \text{ microsec.} \\ &= 28 \text{ MIPS} \end{aligned}$$

Conclusion

There are a number of ways in which the processing rate of a data-flow processor can be extended. First, the size of the Instruction Memory and the number of Operation Units can be increased. If the additional Cells are fully utilized, the processing rate will grow linearly with the number of Cells added. Second, the bottlenecks of the machine, the output of the Arbitration Network and the input of the Distribution Network could be fabricated in a faster technology. A change from TTL to ECL at the bottlenecks should allow a five-fold increase in the processing rate. Naturally, the slower portions of the networks must be structured in more parallel forms to maintain this rate. A technology change would also allow a decrease in the number of Operation Units if they were to be constructed of the faster technology.

References

[1] Adams, D. A., A Computation Model With Data Flow Sequencing, School of Humanities and Sciences, Stanford University, Stanford, Calif., (December, 1968).

[2] Bährs, A., "Operation Patterns (An Extensible Model of an Extensible Language)," Symposium on Theoretical Programming, Novosibirsk, USSR, (August, 1972).

[3] Dennis, J. B., "First Version of a Data Flow Procedure Language," Lecture Notes in Computer Science 19, (G. Goos and J. Hartmanis, Eds.), Springer-Verlag, New York (1974), pp. 362-376.

[4] Dennis, J. B., and D. P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," Proceedings of the ACM 1974 National Conference, ACM, New York, (November, 1974), pp. 402-409.

[5] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, (January, 1975), pp. 126-132.

[6] Karp, R. M., and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," SIAM Journal of Applied Mathematics 14 (November, 1966), pp. 1390-1411.

[7] Kosinski, P. R., "A Data Flow Language for Operating Systems Programming," Proceedings of the ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, (September, 1973), pp. 89-94.

[8] Misunas, D. P., A Computer Architecture for Data-Flow Computation, SM Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., (June, 1975).

[9] Misunas, D. P., "Structure Processing in a Data-Flow Computer," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, (August, 1975), pp. 230-234.

[10] Rodriguez, J. E., A Graph Model for Parallel Computation, Report TR-64, Project MAC, M.I.T., Cambridge, Mass., (September, 1969).

[11] Rumbaugh, J. E., A Parallel Asynchronous Computer Architecture for Data Flow Programs, Report TR-150, Project MAC, M.I.T., Cambridge, Mass. (May, 1975).

[12] Rumbaugh, J. E., "A Data Flow Multiprocessor," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York (August, 1975), pp. 220-223.

AN ANALYTIC APPROACH TO PERFORMANCE ANALYSIS FOR A CLASS OF DATA FLOW PROCESSORS[†]

by

Susan C. Meyer
Department of Mathematics
Clarkson College of Technology
Potsdam, New York 13676

Abstract -- In this paper, an analytic method is given for determining the projected performance of a restricted class of data flow processors. We present a model for describing processes as implemented on these systems which is similar to Karp-Miller computation graphs. Using this model, we derive bounds on the time required for an implemented computational process. Since this performance measure is highly dependent on the assignment of operations in the process to functional units, we also investigate the topic of operation assignment. Time optimal assignments are defined as those which impose no artificial restrictions on the time performance of the implemented process, and conditions are derived under which a given operator assignment is time optimal.

I. Introduction

In this paper we develop a method for determining the time required by programs implemented on a class of data flow processors. In doing so, we provide an analytic approach to estimating the projected performance of data flow processors.

Data flow processes are those in which each operation is allowed to occur whenever all of its operands are available, irrespective of any external timing considerations. In this paper we investigate data flow programs which can be modeled by Karp-Miller computation graphs [1] and their implementation on processors which are capable of realizing them. Various architectures have been proposed for the class of data flow processors which we consider [2,3]. The techniques developed in the paper are general in nature and are applicable in the context of these proposed architectures.

The Karp-Miller computation graph is a model for parallel computation in which each vertex of a directed graph represents an operation and each edge of the graph is viewed as a queue which may contain data. Performance of an operation causes data to be removed from its input queues and results placed on its output queues. An operation may occur only if there are a sufficient number of operands available on each of its input queues.

The class of processes which can be described using this model is restricted to those which involve no data dependent branching. Though this restriction is a severe one, there are compelling reasons for studying the performance of these processes as implemented on data flow processors. Among these reasons are the probability that the first data flow processors to be built will fall into the class which realizes just this type of computation. It is also true that most data flow programs are composed primarily of segments that can be described under the restrictions indicated.

In the following section we give a formal definition of the Karp-Miller model and describe some of its properties. Since we are concerned with data flow processes as implemented on data flow processors, we have modified the Karp-Miller model to represent factors which affect processor performance. This modified model is also presented in the next section. In the third section we show how to calculate the time required by an implemented data flow program, and in the fourth we give necessary and sufficient conditions for time optimality.

II. Foundations

The processes whose implementation dependent characteristics we investigate in this paper are those which can be modeled by Karp-Miller computation graphs. These graphs represent parallel computation by associating an operation with each vertex and viewing each edge as a queue which may contain data. The initial distribution of data and parameters governing queue operation are also specified in the model.

<u>Definition 2.1</u> A Karp-Miller computation graph is a quadruple $C = (V,E,\mu_0,\varphi)$, where:

1) V is the finite vertex set and
2) $E \subseteq V\text{x}V$ is the edge set of a directed graph,
3) μ_0 is a function from E to Z^+ called the <u>initial marking</u>,[(a)]
4) φ is a function, called the <u>firing function</u>, from E to $Z^+\text{x}Z^+\text{x}Z^+$ such that if $\varphi(e) = (i,j,k)$ then $j \geq k$.

The dynamic behavior of the modeled computation is specified by the firing function. This function is usually written as three functions: φ_i (the edge input function), φ_t (the threshold function) and φ_o (the edge output function) where $\varphi(e) = (\varphi_i(e),\varphi_t(e),\varphi_o(e))$ for all edges e in E. For an operation to occur, there must be at least $\varphi_t(e)$ data items on each of its input queues, e. When that operation occurs, $\varphi_o(e)$ items are removed from each of its input queues, e, and $\varphi_i(e')$ items are placed on each of its output queues, e'. In this paper we are only concerned with implementations of Karp-Miller computation graphs in which $\varphi_i(e)$ and $\varphi_o(e)$ are strictly positive. Such graphs are said to be <u>productive</u>.

An example of a Karp-Miller computation graph

[†]This work was supported by the National Science Foundation under grant MCS76-07681.

[(a)] Z^+ denotes $\{0,1,2,\ldots\}$, the set of nonnegative integers.

is shown in Figure 1. In this graph, the firing function φ is given by a vector of numbers associated with each edge. The initial distribution of data is indicated by the presence of $\mu_0(e)$ darkened circles on each edge.

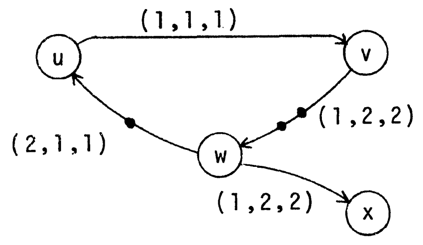


Figure 1

Since we are concerned with the processes described by Karp-Miller computation graphs as implemented on data flow processors, we modify the Karp-Miller model to represent implementation dependent factors. One major modification is necessary because there will almost never be as many functional units of a given type available in the processor as there are operations of that type in the modeled computation. For this reason, an assignment of operations in a computation graph to functional units which realize them must be made. This is done by means of an operator assignment function.

Another restriction which is imposed in implementation is that the queue lengths must be bounded. This restriction is incorporated by adding a bounding component, φ_b, to the firing function φ. Then an operation may occur only if there is sufficient data available <u>and</u> no output queue of that operation overflows as a result of that operation's occurrence.

Finally, since we are concerned with determining bounds on the time required for a process as implemented, we must introduce a timing function to provide a bound on the time required for each functional unit to complete an operation. (A similar function has been used in references [4,5]). Thus, we have the following model of data flow computations as implemented on a data flow processor.

<u>Definition 2.2</u> An <u>implemented computation graph</u> (ICG) is a seven-tuple $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$, where:

1) V is the countable vertex set and
2) $E \subseteq V\times V$ is the edge set of a directed graph,
3) O is a finite set of <u>operators</u>, or functional units,
4) μ_0 is a function from E to Z^+ called the <u>initial marking</u>,
5) φ is a function, called the <u>firing function</u>, from E to $Z^+ \times Z^+ \times Z^+ \times Z^+$ such that if $\varphi(e) = (i,j,k,\ell)$, $\ell \geq j \geq k$ and $\ell \geq i$,
6) α is a (total) function from V to O called the <u>operator assignment function</u>, and
7) τ is a function, called the <u>timing function</u>, from O to R^+ which gives the minimum time required for each operator to complete an activation.[(b)]

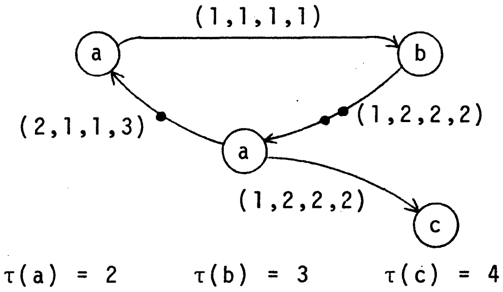
The firing function is usually expressed in a similar way as that for the Karp-Miller model, as four functions. Thus, $\varphi(e) = (\varphi_i(e),\varphi_t(e),\varphi_o(e),\varphi_b(e))$ for all edges e in E, where φ_i, φ_t, and φ_o are as before and φ_b is the queue bounding function. Since we are only concerned with implementations of productive Karp-Miller graphs in this paper, the firing function of our implemented computation graphs will have range NxNxNxN.[(c)]

A distribution of data items in the graph, represented by a function, μ, from E to Z^+, is called a <u>marking</u>. New items are produced and used by the system according to the specification given by the firing function φ. If μ is a marking, then a queue e contains $\mu(e)$ items under that marking. A vertex v in V is <u>firable</u> whenever $\mu(e) \geq \varphi_t(e)$ for all edges e directed into v and for all edges e' directed out of v, $\mu(e) + \varphi_i(e') \geq \varphi_b(e')$. When v occurs, or fires, $\varphi_o(e)$ items are removed from each edge directed into v and $\varphi_i(e')$ items are placed on each edge e' directed out of v. Thus, a new marking μ' is produced, where:

$$\mu'(e) = \begin{cases} \mu(e)+\varphi_i(e) & \text{if e is directed out of but not into v,} \\ \mu(e)-\varphi_o(e) & \text{if e is directed into but not out of v,} \\ \mu(e)+\varphi_i(e)-\varphi_o(e) & \text{if e is directed into and out of v, and} \\ \mu(e) & \text{otherwise.} \end{cases}$$

The firing of a vertex models the occurrence of an operation, and the <u>behavior</u> of the system is the set of all sequences of legal operation occurrences. The function α associates the vertices with operators used to realize these operations. When two vertices are assigned the same operator ($\alpha(u) = \alpha(v)$), the firing of these two vertices represent two different initiations of the same operator.

An implementation of the Karp-Miller computation graph shown in Figure 1 is illustrated by the ICG of Figure 2. In this ICG, the operator assignment function (and operator set) are given by the vertex labeling in the graph.



$\tau(a) = 2 \qquad \tau(b) = 3 \qquad \tau(c) = 4$

Figure 2

(b) R^+ denotes the positive real numbers.

(c) N denotes {1,2,3,...}, the set of natural numbers.

In order to investigate the timing characteristics of an ICG, it is convenient to develop a representation for its behavior which explicitly shows the constraints on each occurrence of each operator. The infinite structure defined below is such a representation.

Definition 2.3 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG. The behavior graph of C is the (unbounded) ICG. $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ given by:

1) $\underline{V} = \{x(v)^i | \alpha(v)=x \text{ for } v\in V \text{ and } i\in Z\}$,

2) $\underline{E}$: $e = (x(u)^i,y(v)^j)\in\underline{E}$ whenever:

   i) $u = v$ and $j = i + 1$, or

   ii) there is an edge $e = (u,v)$ in E such that $x=\alpha(u)$, $y=\alpha(v)$,

   $$\text{and } i=\left\lceil\frac{(j-1)\varphi_o(e)-\mu_0(e)+\varphi_t(e)}{\varphi_i(e)}\right\rceil, \text{or}$$

   iii) there is an edge $e = (v,u)$ in E such that $x=\alpha(u)$, $y=\alpha(v)$,

   $$\text{and } i=\left\lceil\frac{j\varphi_i(e)+\mu_0(e)-\varphi_b(e)}{\varphi_o(e)}\right\rceil \quad (d),$$

3) $\underline{O} = O$,

4) for $e = (x(u)^i, y(v)^j)$ in $\underline{E}$,

$$\underline{\mu}_0(e) = \begin{cases} 1 \text{ if } i\leq) \text{ and } j>0 \text{ or } i>0 \text{ and } j\leq 0 \\ 0 \text{ otherwise,} \end{cases}$$

5) $\underline{\varphi}(e) = (1,1,1)$ for each edge e in $\underline{E}$,

6) $\underline{\alpha}(v) = x$ for all $v = x(u)^i$ in $\underline{V}$, and

7) $\underline{\tau} = \tau$.

Notice that the firing function for the behavior graph of an ICG contains no queue bounding function. Such a function is not necessary because queue lengths in this infinite graph are already bounded. The behavior graph of an ICG is also an infinite marked graph [6,7] (when the timing function is disregarded). A portion of the behavior graph for the ICG of Figure 2 is shown in Figure 3.

Proposition 2.1 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG and let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ be its behavior graph. Then C and B_C have the same behavior.

A rigorous proof of this result is straightforward but tedious. The interested reader can find a proof for the case in which α is one-to-one in [8]. Similar arguments establish the result in the more general case.

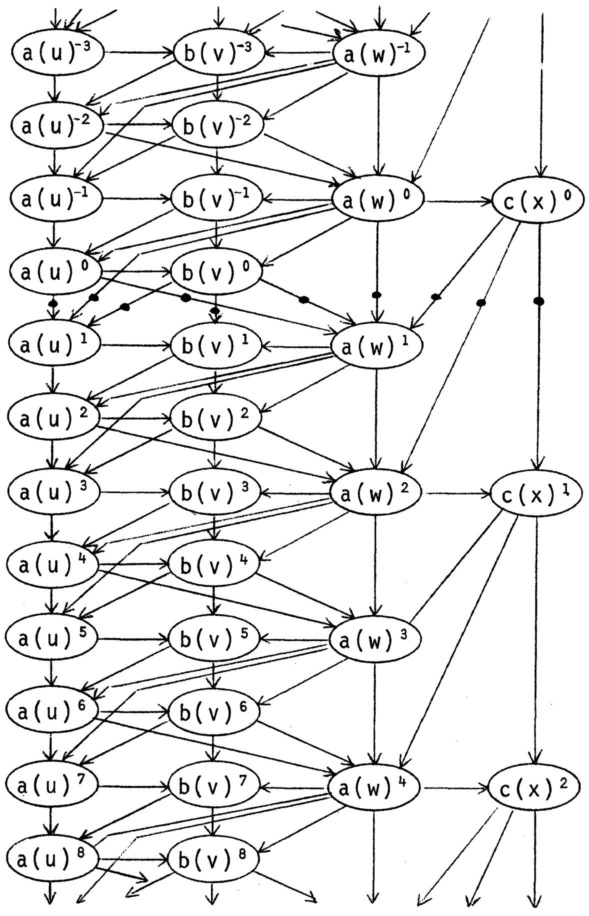


Figure 3

In examining the timing characteristics of these implemented computation graphs, we must consider one further constraint which is imposed on such a system by physical limitations of the devices realizing it. These constraints can be illustrated with the aid of the behavior graph shown in Figure 3. It is clear, on examination of this graph, that the vertices $a(u)^1$ and $a(w)^1$ may be concurrently enabled and that they are assigned to the same functional unit. It is usually required that a functional unit complete one activation before it may begin another. Therefore, one or the other of these operations must be performed first.

The physical limitation that only one activation of a functional unit may be in execution at a time places a total ordering on the operations assigned to a given operator. Such an ordering indexes the activations of operators in the system and can be specified in the following way.

(d) $\lceil x\rceil$ denotes the ceiling of x. Thus, $\lceil x\rceil$ is the least integer greater than or equal to x.

Definition 2.4 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG and let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ be its behavior graph. An ordering for operator a (where $a \epsilon O$) is a one-to-one function, ω_a, from $\underline{\alpha}^{-1}(a)$ to Z [(e)] such that $\omega_a(a(u)^i) > 0$ whenever $i > 0$ and $\omega_a(a(u)^i) \leq 0$ whenever $i \leq 0$ for all $a(u)^i \epsilon \underline{\alpha}^{-1}(a)$. An operator ordering function for C is a function $\omega = \bigcup_{a \epsilon \underline{O}} \omega_a$, where all of the subfunctions ω_a are orderings for operators in $\underline{O}$.

The constraints imposed by an operator ordering function ω may be represented in the behavior graph of an ICG by adding edges from vertex u to vertex v whenever $\alpha^{-1}(u) = \alpha^{-1}(v) = a$ and $\omega_a(u)+1 = \omega_a(v)$ for $\omega_a \subseteq \omega$. Thus, each operator ordering gives rise to a new infinite (unbounded) ICG defined below.

Definition 2.5 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG and let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ be its behavior graph. Let ω be an operator ordering function for C. The ω-constrained behavior graph $B_C(\omega) = (V^\omega,E^\omega,O^\omega,\mu_0^\omega,\varphi^\omega,\alpha^\omega,\tau^\omega)$ is given by

1) $V^\omega = \underline{V}$,
2) $E^\omega = \underline{E} \cup \{(u,v) \mid \underline{\alpha}(u)=\underline{\alpha}(v) \text{ and } \omega(u)+1 = \omega(v)\}$,
3) $O^\omega = \underline{O}$,
4) $$\mu_0^\omega(e)=\begin{cases} 1 & \text{if } e=(x(u)^i,y(v)^j) \text{ and either } i\leq 0 \text{ and } j>0 \text{ or } j\leq 0 \text{ and } i>0 \\ 0 & \text{otherwise,}\end{cases}$$
5) $\varphi^\omega(e) = (1,1,1)$ for all e in $\underline{E}$,
6) $\alpha^\omega = \underline{\alpha}$, and
7) $\tau^\omega = \underline{\tau}$.

Clearly, some of the operator ordering functions which are well defined are not consistent with the natural constraints already present in the system. For example, in Figure 3, if we let $\omega_a(a(u)^1) = 1$, $\omega_a(a(u)^2) = 2$, and $\omega_a(a(w)^1) = 3$, a set of constraints which causes the process to stop prematurely has been created. This is not a desirable situation, and if the system is simply allowed to run freely, such an ordering will never be chosen.

We denote the number of times an operator a in an ICG C (or its behavior graph B_C) by $\#(a|C)$ (or $\#(a|B_C)$). Then an operator ordering function ω is said to be legal if $\#(a|B_C) = \#(a|B_C(\omega))$ for each operator a in $\underline{O}$. The only operator functions considered in this paper are legal ones. Figure 4 shows a portion of $B_C(\omega)$ for a legal operator ordering ω. The heavy lines have been inserted to enforce ω.

[(e)] Z denotes $\{\ldots-2,-1,0,1,2\ldots\}$, the set of integers.

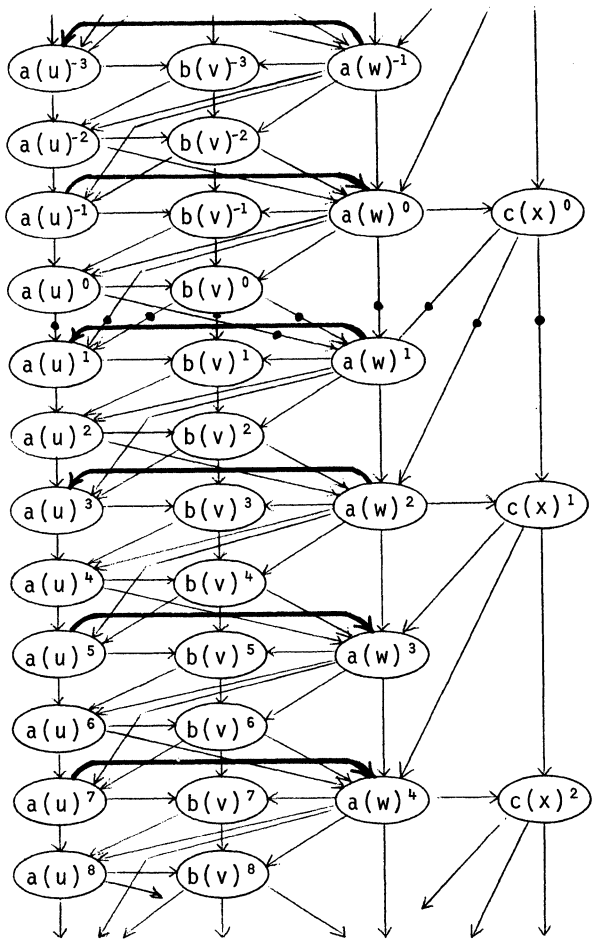


Figure 4

Remark. It is not difficult to see how legal operator ordering functions can be constructed. One only need find all topological sorts of the partial order on $\alpha^{-1}(a)$ implicit in the behavior graph B_C for each operator a in $\underline{O}$. Call this set T_a. Then by selecting one element from each of the sets T_a to obtain a set of ω_a which are compatible, one constructs a legal operator ordering function.

When the modeled process is carried out as implemented, it is not known which of the legal operator orderings actually is chosen. All of them are feasible and may occur. An analysis of timing characteristics must therefore consider all possible legal operator orderings.

Proposition 2.2 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG. Then C can behave in one of two ways:

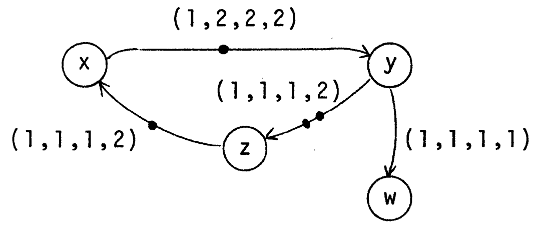
1) it may fire several vertices and then reach a marking under which no vertex is firable, or

2) it may fire several vertices, reaching a previously observed marking, after which the behavior is repetitive.

A proof of this result is somewhat lengthy. The interested reader may find a proof of a similar result in reference [8].

III. Timing Characteristics

In this section we first study timing characteristics of ICG's which fire several vertices then reach a marking under which no vertex is firable. Such implemented graphs are relatively straightforward to analyze because the behavior of the system is a finite set. Figure 5 gives an example of such an ICG, and part of its behavior graph is shown in Figure 6. Notice that in Figure 6, there is a path from $b(y)^4$ to $a(z)^6$ to $a(x)^7$ to $b(y)^4$ to $b(w)^4$. The presence of this path indicates that b(y) and b(w) can occur no more than three times each, a(z) can occur at most five times, and a(x) may occur six times.



$\alpha(x) = a$

$\alpha(y) = b \qquad \tau(a) = 3$

$\alpha(z) = a \qquad \tau(b) = 4$

$\alpha(w) = b$

Figure 5

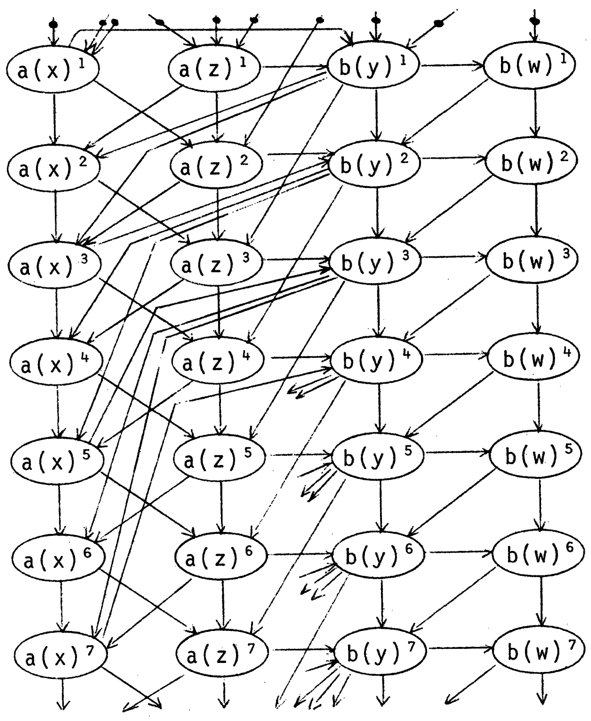


Figure 6

<u>Proposition 3.1</u> Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG with finite behavior. There is a finite ICG, M_C, which has the same behavior as C in which $\varphi_i(e) = \varphi_o(e) = \varphi_t(e) = 1$ for each edge.

<u>Proof</u>. The proof is by construction. Let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ be the behavior graph for C and let $\#(v|C)$ be the maximum number of times a vertex v occurs in C. The $M_C = (V',E',O',\mu_0',\varphi',\alpha',\tau')$ is given by:

1) $V'=\{x(v)^i \epsilon \underline{V} | 0 \le i \le \#(v|C)\}$,

2) $E'=\{(u,v)\epsilon\underline{E} | u,v\epsilon V'\} \cup \{(v^0,v^0) | v^i = x(u)^i \epsilon V'\}$

3) $O'=\underline{O}$,

4) $\mu_0(e)=\begin{cases}1 & \text{if } e=(x^0,y^1) \text{ in } E' \\ 0 & \text{otherwise}\end{cases}$

5) $\varphi'(e)=(1,1,1,2)$ for all e in E',

6) $\alpha'(v)=\alpha(v)$ for all vertices $v\epsilon V'$, and

7) $\tau'=\tau$.

It is readily established that the new graph M_C has the same behavior as B_C, hence the same as that of C as well. $\square$

Given this finite ICG, we can calculate a lower bound on the time <u>required</u> to complete the computation, given that a particular legal operator ordering, ω, has been chosen. We denote the minimum time required for completion under ω by ρ_ω and compute ρ_ω with the aid of the following ω-constrained finite ICG.

<u>Definition 3.1</u> Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG with finite behavior and let $M_C = (V',E',O',\mu_0',\varphi',\alpha',\tau')$ be defined as above. Then $M_C(\omega) = (V^\omega,E^\omega,O^\omega,\mu_0^\omega,\varphi^\omega,\alpha^\omega,\tau^\omega)$, for a legal operator ordering ω, is given by:

1) $V^\omega=V'$,

2) $E^\omega=E'\cup\{(u,v) | u,v\epsilon V^\omega, \alpha'(u)=\alpha'(v), \text{ and } \omega(u)+1 = \omega(v)\}$,

3) $O^\omega=O'$,

4) $\mu_0^\omega(e)=\begin{cases}1 & \text{if } e=(x^0,y^1) \\ 0 & \text{otherwise}\end{cases}$

5) $\varphi^\omega(e)=(1,1,1,2)$ for all e in E^ω ,

6) $\alpha^\omega = \alpha'$, and

7) $\tau^\omega=\tau'$.

Figure 7 shows $M_C(\omega)$ for the ICG of Figure 5 and a legal operator ordering ω.

The minimum time required to complete the process under the legal operator ordering, ω, can now be readily determined. Let $\iota=\{v\epsilon V^\omega | v \text{ is firable under } \mu_0^\omega\}$ and let Π be the set of all paths $\pi=v_1v_2...v_n$ in $M_C(\omega)$ such that $v_1\epsilon\iota$. Then we have the following result.

<u>Proposition 3.2</u> Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG with finite behavior. Let ω be a legal operator ordering for C and $M_C(\omega)$ be the ω-constrained graph defined above. Then $\rho_\omega = \max_{\pi\epsilon\Pi}(\sum_{v\epsilon\pi}\tau^\omega(v))$.

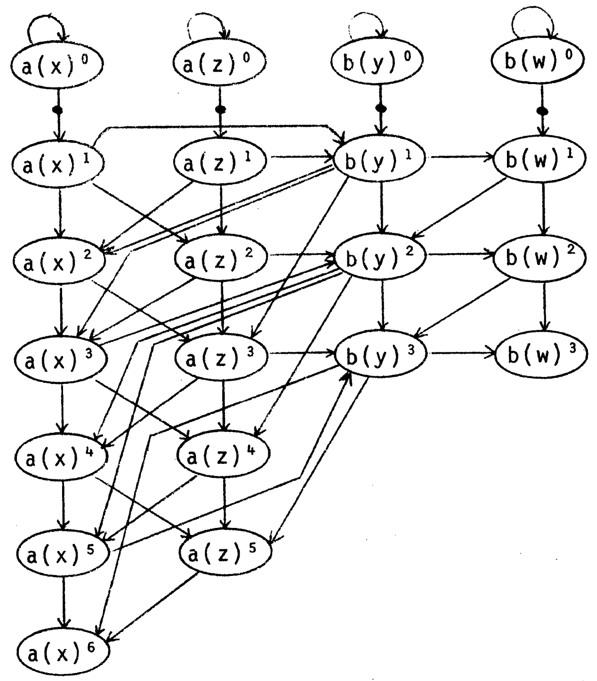


Figure 7

Unfortunately, it is not known which of the legal operator orderings for C is chosen each time the computation is performed. Thus, there are two figures of interest to us:

1) a time, ρ, which is the minimum time in which the computation may be completed when the operator ordering chosen is not known, and

2) a time, ρ_{min}, which is the minimum possible time for completion of the process.

Although the behavior of C is finite, the number of legal operator orderings for C is still infinite. However, the only part of a legal operator ordering ω which is of any consequence in this case is the restriction of ω to vertices v which are also in M_C. This allows us to restrict our attention to the finite set, W(C), of legal operator orderings for M_C. The two timing figures of interest can be calculated according to the following proposition.

Proposition 3.3 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG with finite behavior and let W(C) be the set of legal operator orderings for M_C. Then

$$\rho = \max_{\omega\in W(c)} (\rho_\omega) \quad \text{and} \quad \rho_{min} = \min_{\omega\in W(C)} (\rho_\omega) \quad .$$

For the example shown in Figure 5, $\rho = 42$ and $\rho_{min} = 39$.

The analysis of nonterminating graphs is not quite so simple as that of terminating ones. In this case the behavior is infinite, so there are an infinite number of legal orderings which must be considered. Furthermore, the concept of "time required for completion" has no meaning for non-terminating graphs. For this reason we use the concept of computation rate to obtain a measure of system performance. The computation rate of the implemented process is defined as the number of operator occurrences per unit time. It is clearly desirable to maximize this ratio if at all possible.

If an ICG does not terminate, there are two classes of behavior it can exhibit. If $\mu_0(e)<\varphi_t(e)-\varphi_o(e)$ for some edge e, the graph will have an initial transient behavior followed by a cyclic steady state behavior. If each edge e has $\mu_0(e)\geq\varphi_t(e)-\varphi_o(e)$, there is no transient behavior and the graph is said to be repetitive. Once a vertex has fired for the first time, there will always be at least $\varphi_t(e)-\varphi_o(e)$ items on an edge, so nonterminating ICG's eventually become repetitive. Since we are concerned with calculating the computation rate of an ICG, the transient behavior (if any) is of no concern to us. We therefore, assume that all of the nonterminating ICG's considered are repetitive.

In order to deal with the difficulty which arises because there are infinitely many legal operator orderings to consider, we must characterize the structure of an ICG's behavior. This has been done for structures very similar to ICG's in [8], and the results in that case closely parallel those needed here. The interested reader may refer to [8] for rigorous proofs.

Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG and let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ be its behavior graph. There is always a finite portion of B_C which generates the infinite graph in the sense that B_C is made up of infinitely many copies of that finite subgraph. These concepts are formalized with the aid of the following definitions.

Definition 3.2 Let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ be the behavior graph of an ICG $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$. Let Ψ be a function from V to N. Then

$$\underline{E}/\Psi = \{(x(u)^i,y(v)^j)\in\underline{E} \mid 1\leq i\leq\Psi(u)\} \quad .$$

The set generated by Ψ is then given by:

$$<\underline{E}/\Psi>=\{(x(u)^{i+n\Psi(u)},y(v)^{j+n\Psi(v)}) \mid (x(u)^i,y(v)^j) \in\underline{E}/\Psi \text{ and } n\in Z\} \quad .$$

If $<\underline{E}/\Psi>=\underline{E}$, then Ψ is a generator for B_C .

Proposition 3.4 Let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\varphi,\alpha,\tau)$. Then there is always a generator Ψ for B_C.

Proof. Since C does not terminate, we can always find a positive integer solution to the set of equations:

$$\{z_m = \frac{\varphi_o(e)}{\varphi_i(e)} z_n \mid e = (m,n)\in E\} \quad .$$

Let $\{z_m | m\epsilon V\}$ be a positive integer solution for this set of equations, and let $\Psi(\alpha(v)(v))=z_v$ for each $v\epsilon V$. It is readily established that Ψ is a generator for B_C. □

The function Ψ, where $\Psi(u) = 4$, $\Psi(v) = 4$, $\Psi(w) = 2$, and $\Psi(x) = 1$ is a generator for the graph B_C of Figure 3.

Now let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ be the behavior graph for a nonterminating ICG $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ and let Ψ be a generator for B_C. We denote the (finite) set of all legal operator orderings of the vertices in the set $\{x(u)^i | 1\le i\le\Psi(u)\}$ by $W^0(C,\Psi)$. The key to dealing with the problem of considering an infinite number of legal operator orderings lies in showing that the infinite set of legal operator orderings for C may be "generated" by this set $W^0(C,\Psi)$.

Let ω be a legal operator ordering for C and consider the graph $B_C(\omega)$. This graph is identical to B_C except that edges have been added to B_C which enforce the ordering ω. Thus, we may write:

$$E^\omega = <\underline{E}/\Psi> \cup \{(x(u)^i, y(v)^j) | x=y \text{ and } \omega(x(u))+1=\omega(y(v))\} .$$

Now consider the set of edges

$$(\underline{E}/\Psi)^n = \{(x(u)^{i+n\Psi(u)}, y(v)^{j+n\Psi(v)})\epsilon E^\omega | (x(u)^i, y(v)^j)\epsilon\underline{E}/\Psi\} .$$

Notice that $(\underline{E}/\Psi)^n$ is the n+1st copy of $\underline{E}/\Psi$ in E^ω and let ω^n be the restriction of ω to the set $\underline{V}^n=\{x(u)^{i+n\Psi} | (x(u)^{i+n\Psi(u)}, y(v)^{j+n\Psi(v)})\epsilon(\underline{E}/\Psi)^n\}$. Since ω is a legal operator ordering for C, ω^n must be a legal operator ordering for vertices in $\underline{V}^n$. This can be true if and only if there is a legal ordering $\omega°$ for vertices in $\underline{V}^0$ such that $\omega°(x(u)^i))=\omega^n(x(u)^{i+n\Psi(u)})-(n-1)[\sum_{u\epsilon\alpha^{-1}(x)}\Psi(u)]$ for each vertex $x(u)^i\epsilon\underline{V}^0$.

Let $(\omega°)^n(x(u)^{i+n\Psi(u)}) =$

$$\omega°(x(u)^i)+(n-1)[\sum_{u\epsilon\alpha^{-1}(x)}\Psi(u)] .$$

It has just been established that for <u>any</u> legal operator ordering ω, we may write $\omega^n=(\omega°)^n$ for some $\omega°\epsilon W^0(C,\Psi)$. Thus, we may write:

$$\omega = \bigcup_{-\infty\le n\le\infty}(\omega^n) = \bigcup_{-\infty\le n\le\infty}((\omega°_n)^n | \omega°_n\epsilon W^0(C,\Psi)) .$$

$W^0(C,\Psi)$ is a finite set, so it has been shown that both the graph B_C and the infinite set of legal operator orderings can be finitely generated.

Now let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG and let ω be a legal operator ordering for C. Recall that the computation rate of an ICG is the number of operation occurrences per unit time. Since the graph $B_C(\omega)$ is finitely generated, we may write the following expression for ρ_ω, the maximum computation rate of C under ordering ω:

$$\rho_\omega = \lim_{i\to\infty}\frac{i\sum_{u\epsilon V}\Psi(u)}{T_i(\omega)}$$

where $T_i(\omega)$ is the time required to complete the execution of vertices in $\{v\epsilon\underline{V}^n | \le n\le i-1\}$ and $\Psi(u)$ is any generator for $B_C(\omega)$. It is easy to calculate $\sum_{u\epsilon V}\Psi(u)$ for any generator Ψ, so we need only find an expression for $T_i(\omega)$ to determine ρ_ω.

<u>Notation</u>. If Ψ is a generator for B_C, we let $\iota_i=\{v\epsilon V^0 | v \text{ is firable under } \mu_0 \text{ and } \omega_i\epsilon W^0(C,\Psi)\}$, and let $\Pi(\omega_i)=\{\pi=v_1\ldots v_n | v_i\epsilon\underline{V}^0 \text{ for } i=1,2,\ldots,n,$ $v_1\epsilon\iota_i$, and $(v_i,v_{i+1})\epsilon E^\omega$.

<u>Proposition 3.5</u> Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG and B_C be the behavior graph for C. There is a generator Ψ for B_C such that for each ordering $\omega_i\epsilon W^0(C,\Psi)$ and every path $\pi=v_1,\ldots v_n$ in $\Pi(\omega_i)$, there is an edge $(v_n,v_1^{1+\Psi(u)})$ in $\underline{E}/\Psi$, where $v_1 = x(u)^1$.

To construct a generator satisfying the conditions of Proposition 3.5, one begins with the minimal generator Ψ_{min}. If Ψ_{min} does not have the required structure, then $k*\Psi_{min}$ does for some integer k, and $k*\Psi_{min}$ is also a generator for B_C.

Now let Ψ be a generator for B_C which satisfies the conditions of Proposition 3.5. Then we may write:

$$T_i(\omega) = \sum_{n=0}^{i-1} T(\omega^n)$$

where $T(\omega^n)$ is the time required to complete only the nth copy of $\underline{E}/\Psi$ in $B_C(\omega)$. Thus we have:

$$\rho_\omega = \lim_{i\to\infty}\frac{i\sum_{u\epsilon V}\Psi(u)}{\sum_{i=1}^{n} T(\omega^n)}$$

where Ψ is any generator for $B_C(\omega)$ satisfying the conditions of Proposition 3.5.

Since each ω^n is generated by some $\omega_n\epsilon W^0(C,\Psi)$, we know that $T(\omega^n) = \max_{\pi\epsilon\Pi(\omega_n)}(\sum_{v\epsilon\pi}\tau(v))$, where ω_n generates ω^n. Given this expression for $T(\omega^n)$, we may derive bounds on computation rates for nonterminating ICG's.

As before, there are two figures which are of interest to us:

1) a rate, ρ, which is the maximum feasible rate given that the operator ordering chosen is not known, and
2) a rate, ρ_{min}, which is the maximum possible rate that can be achieved.

<u>Proposition 3.6</u> Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG and let ω be a legal operator ordering for C. Then

$$\frac{\sum_{u\epsilon V}\Psi(u)}{T_{max}} \le \rho_\omega \le \frac{\sum_{u\epsilon V}\Psi(u)}{T_{min}} \text{ , where:}$$

1) Ψ is a generator for the behavior graph, B_C, of C satisfying the conditions of Proposition 3.5,

2) $T_{max} = \max_{\omega_i \in W^0(C,\Psi)} (\max_{\pi\in\Pi(\omega_i)} (\Sigma_{v\in\pi} \tau(v)))$, and

3) $T_{min} = \min_{\omega_i \in W^0(C,\Psi)} (\max_{\pi\in\Pi} \tau(v)))$.

Proof. Notice that $T_{min} \le T(\omega^n) \le T_{max}$ for every possible choice of ω^n. Thus, we may write:

$$iT_{min} \le \sum_{n=1}^{i} T(\omega^n) \le iT_{max}$$

Substituting, then, we have:

$$\rho_\omega = \lim_{i\to\infty} \frac{i\sum_{u\in V}\Psi(u)}{\sum_{n=1}^{i} T(\omega^n)} \ge \lim_{i\to\infty} \frac{i\sum_{u\in V}\Psi(u)}{iT_{max}} = \frac{\sum_{u\in V}\Psi(u)}{T_{max}}$$

and

$$\rho_\omega = \lim_{i\to\infty} \frac{i\sum_{u\in V}\Psi(u)}{\sum_{n=1}^{i} T(\omega^n)} \le \lim_{i\to\infty} \frac{i\sum_{u\in V}\Psi(u)}{iT_{min}} = \frac{\sum_{u\in V}\Psi(u)}{T_{min}} \quad \square$$

Corollary 3.1 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG with T_{max}, T_{min}, and Ψ as above. Then

$$\rho = \frac{\sum_{u\in V}\Psi(u)}{T_{max}} \quad \text{and} \quad \rho_{min} = \frac{\sum_{u\in V}\Psi(u)}{T_{min}} \quad .$$

For the ICG of Figure 2, ρ=11/24 and ρ_{min} = 11/24.

IV. Time Optimality

In this section we derive necessary and sufficient conditions under which no penalty is imposed on the rate of computation due to operator assignment. Throughout the remainder of the paper, we shall assume that Ψ is a generator for the behavior graph which satisfies the conditions of Proposition 3.5. Furthermore, since it is not known which of the many legal operator orderings will be chosen, we concentrate on the rate ρ rather than ρ_{min}. We present results for the case in which a computation graph (hence the corresponding ICG) is nonterminating. The analysis for the terminating case is similar. It is first shown that the maximum computation rate, ρ, of a nonterminating ICG cannot exceed that of its one-to-one equivalent.

Definition 4.1 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG. Then the one-to-one equivalent to C is an ICG $C'=(V',E',O',\mu_0',\varphi',\alpha',\tau')$ given by:

1) $V' = V$,
2) $E' = E$,
3) $O' = V' = V$,
4) $\mu_0' = \mu_0$,
5) $\varphi' = \varphi$,
6) $\alpha'(v) = v$ for all $v\in V'$, and
7) $\tau'(v) = \tau(a)$, where $\alpha(v) = a\in O$, for each $v\in O'$.

Notice that C differs from C only in its operator assignment function (and operator set). To allow a meaningful comparison of computation rates, the function τ' is defined so that if $\alpha(u)=\alpha(v)$ then $\tau(u)=\tau(v)$ as well. This allows us to isolate the effects of operator assignment alone. Figure 8 shows the one-to-one equivalent to the ICG of Figure 2.

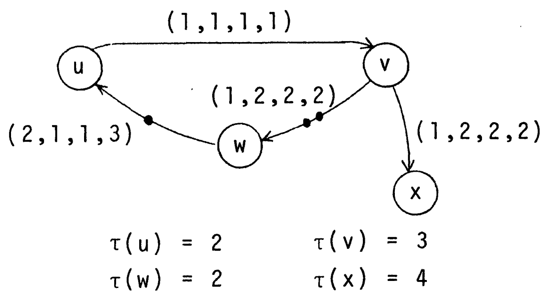


Figure 8

Proposition 4.1 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG and $C' = (V',E',O',\mu_0',\varphi',\alpha',\tau')$ be its one-to-one equivalent. Let $\rho(C)$ and $\rho(C')$ denote the computation rates of C and C', respectively. Then $\rho(C)\le\rho(C')$.

Proof. Notice that the behavior graphs B_C and $B_{C'}$ for C and C' are isomorphic, differing only in their vertex labeling.

There is only one legal operator ordering for C' - the one given in $B_{C'}$, but there may be many for C. Let ω be any legal operator ordering for C and consider $B_C(\omega)$. Since B_C is contained in $B_C(\omega)$ and B_C and $B_{C'}$ are isomorphic, it is clear that $T_{max}(C)\ge T_{max}(C')$. Furthermore, it can readily be shown that Ψ is a generator for B_C if and only if Ψ is also a generator for $B_{C'}$. It follows immediately that $\rho(C)\le\rho(C')$. $\square$

This result establishes that no implementation can produce a greater computation rate than can a one-to-one operator assignment. The conditions under which C has at least as great a compuation rate as C' are simply stated and relatively easy, though tedious, to check.

Theorem 4.1 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG and let $C' = (V',E',O',\mu_0',\varphi',\alpha',\tau')$ be its one-to-one equivalent. Then $\rho(C) = \rho(C')$ if and only if $T_{max}(C)=T_{max}(C')$.

Although $W^0(C,\Psi)$ for C is finite, it can be a very large set. Thus, the task of finding $T_{max}(C)$ and $T_{max}(C')$ can be very time consuming. It can be made somewhat less tedious by dealing with a reduced form of the behavior graph which is obtained by eliminating redundant edges. A portion of the reduced behavior graph for the ICG of Figure 2 is shown in Figure 9.

Definition 4.2 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG and let $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ be its behavior graph. The reduced behavior graph for C is an (unbounded) infinite ICG $\underline{\underline{B}}_C=(\underline{\underline{V}},\underline{\underline{E}},\underline{\underline{O}},\underline{\underline{\mu}}_0,\underline{\underline{\varphi}},\underline{\underline{\alpha}},\underline{\underline{\tau}})$, where:

1) $\underline{\underline{V}} = \underline{V}$,
2) $\underline{\underline{E}} = \underline{E}-E_R$ where

$E_R=\{e=(x^i,y^j)\in\underline{E} \mid$ there is a path of length greater than one from x^i to y^j in $\underline{E}$ which has the same initial marking as e},

3) $\underline{\underline{O}} = \underline{O}$,
4) $\underline{\underline{\mu}}_0 = \underline{\mu}_0$ restricted to $\underline{\underline{E}}$,
5) $\underline{\underline{\varphi}} = \underline{\varphi}$ restricted to $\underline{\underline{E}}$,
6) $\underline{\underline{\alpha}} = \underline{\alpha}$, and
7) $\underline{\underline{\tau}} = \underline{\tau}$.

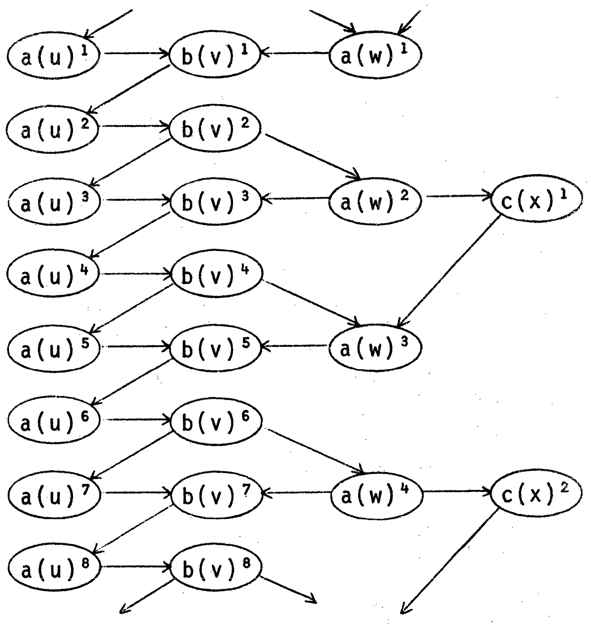


Figure 9

It is easy to see that $\underline{B}_C$ and B_C have the same behavior. All we have done is remove edges which specify redundant constraints.

Notation. Let $C = (V,E,O,\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ and let $\omega_i \epsilon W^0(C,\Psi)$. Then

$$\underline{E}/\Psi(\omega_i) = \underline{E}/\Psi \cup \{(x(u)^i, y(v)^j) \mid x=y \text{ and } \omega_i(x(u)^i) + 1 = \omega_i(y(v)^j)\}$$

and

$$\underline{\underline{E}}/\Psi(\omega_i) = \underline{E}/\Psi(\omega_i) - \{e=(x^i,y^j) \epsilon \underline{E}/\Psi(\omega_i) \mid \text{there is a path of length greater than one from } x^i \text{ to } y^j \text{ in } \underline{E}/\Psi(\omega_i)\}.$$

We also let $\Pi_C(C,\omega_i)$ denote the set of critical paths in B_C under ω_i.

$$\Pi_C(C,\omega_i) = \{\pi \epsilon \Pi(\omega_i) \mid \sum_{v \epsilon \pi} \tau(v) = \max_{\pi' \epsilon \Pi(\omega_i)} (\sum_{v' \epsilon \pi'} \tau(v'))\}.$$

Notice that $\Pi_C(C,\omega_i) \subseteq \underline{E}/\Psi(\omega)$.

Then we have the following result.

Proposition 4.2 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be an ICG with $B_C = (\underline{V},\underline{E},\underline{O},\underline{\mu}_0,\underline{\varphi},\underline{\alpha},\underline{\tau})$ and let $\omega_i \epsilon W^0(C,\Psi)$. Then $\Pi_C(C,\omega_i) \subseteq \underline{\underline{E}}/\Psi(\omega_i)$.

This result indicates that one only need check the sets $\underline{\underline{E}}'/\Psi$ and $\underline{\underline{E}}/\Psi(\omega_i)$ to decide if an operator assignment is time optimal. Thus, the tedium necessary can be reduced, but unfortunately it cannot be eliminated. There are several results which provide sufficient conditions for either time optimality or non-optimality that are quite simple to verify. Unfortunately, these results are not necessary as well as sufficient.

Proposition 4.3 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG and $C' = (V',E',O',\mu_0',\varphi',\alpha',\tau')$ be its one-to-one equivalent with reduced behavior graphs $\underline{B}_C = (\underline{\underline{V}},\underline{\underline{E}},\underline{\underline{O}},\underline{\underline{\mu}}_0,\underline{\underline{\varphi}},\underline{\underline{\alpha}},\underline{\underline{\tau}})$ and $\underline{B}_{C'} = (\underline{\underline{V}}',\underline{\underline{E}}',\underline{\underline{O}}',\underline{\underline{\mu}}_0,\underline{\underline{\varphi}},\underline{\underline{\alpha}},\underline{\underline{\tau}})$, respectively. The operator assignment function α is time optimal if $\underline{\underline{E}}'/\Psi = \underline{\underline{E}}/\Psi(\omega_i)$ for every $\omega_i \epsilon W^0(c,\Psi)$.

Proposition 4.4 Let $C = (V,E,O,\mu_0,\varphi,\alpha,\tau)$ be a nonterminating ICG. The operator assignment function α is time optimal if there is only one legal operator ordering ω.

Remark. Notice that there is a single legal operator ordering ω only when the vertices in $\alpha^{-1}(a)$, for every operator a in an ICG are totally ordered in B_C. In this case, Proposition 4.3 also indicates optimality.

In contrast to the preceding two propositions, the one which follows gives conditions under which an operator assignment function is nonoptimal.

Proposition 4.5 Let $C = (\underline{\underline{V}},\underline{\underline{E}},\underline{\underline{O}},\underline{\underline{\mu}}_0,\underline{\underline{\varphi}},\underline{\underline{\alpha}},\underline{\underline{\tau}})$ and $B_{C'} = (\underline{\underline{V}}',\underline{\underline{E}}',\underline{\underline{O}}',\underline{\underline{\mu}}_0',\underline{\underline{\varphi}}',\underline{\underline{\alpha}}',\underline{\underline{\tau}}')$. If there is some $\omega_i \epsilon W^0(C,\Psi)$ for which $\underline{\underline{E}}/\Psi(\omega_i)$ does not contain every path in $\Pi_C(C',\omega)$ then $\rho(C) > \rho(C')$.

Summary

In this paper we have given a procedure for determining lower bounds on computation rates achievable for data flow programs implemented on machines which can exploit the parallelism inherent in these processes. We have also given conditions under which an operator assignment is time optimal. Unfortunately, these conditions are somewhat tedious to verify, so a number of sufficient conditions for optimality (or non-optimality) have also been given.

Two areas of application for the results presented in this paper immediately come to mind. First, by calculating the maximum computation rate of a data flow program, a hard estimate of the advantage gained in implementing the program on a data flow processor can be obtained. Therefore, these results can provide a basis for an analytic approach to evaluating the projected performance of data flow processors. Secondly, in writing a data flow program, the user of a simulation facility such as the one described by Leung, Misunas, Neczwid and Dennis [2] must effectively specify the operator assignment as part of his program. The results of this paper provide the user of such a system with guidelines for choosing an efficient operator assignment.

References

[1] R.M. Karp and R.E. Miller, Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. of Appl. Math. 14 (November 1966), 1390-1411.

[2] C.K. Leung, D.P. Misunas, A. Neczwid, and J.B. Dennis, A computer simulation facility for packet communication architectures, Proceedings of the Third Annual Symposium on Computer Architecture, IEEE, New York

(January 1976), 58-63.

[3] J.B. Dennis and D.P. Misunas, A computer architecture for highly parallel signal processing. Proceedings of the ACM National Conference, ACM, New York (November 1974).

[4] C. Ramchandani, Analysis of asynchronous concurrent systems by Petri nets, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts (February 1974).

[5] R. Reiter, A study of a model for parallel computations, Ph.D. Thesis, Department of Communication Sciences, University of Michigan, Ann Arbor, Michigan (1967).

[6] F. Commoner, A.W. Holt, S. Even, and A. Pneuli, Marked directed graphs, J. Comput. Syst. Sci. 5 (1971) 511-523.

[7] A.W. Holt and F. Commoner, Events and Conditions, Research Report of Applied Data Research, Inc., New York (1970).

[8] S.C. Meyer, An analysis of two models for parallel computation, Ph.D. Thesis, Department of Electrical Engineering, Rice University, Houston, Texas (December 1974).

ON THE EVALUATION OF ARRAY COMPUTERS

R. Hemmersbach and D. Schütt

Dept. of Computer Science
University of Bonn
53oo Bonn, W-Germany

Summary

A new procedure for measuring and comparing highly parallel computer systems is proposed. Since the behavior of a system is determined by the behavior of its components and the specific modes of combination used, systems are described by sets of labelled acyclic flow graphs (compare [2,8]). The nodes of such graphs represent computer facilities like registers or complete processing elements, the edges characterize the instruction and control flow between the facilities. The labels of nodes and maximal paths ('hyperedges') are induced by the work and power of the facilities and statistical quantities, respectively.

The evaluation of a system is done recursively as follows: Starting with the evaluation of primitive components (first order flow graphs) the results obtained are used for the evaluation of the next level of description (second order flow graphs), i.e. the first order graphs are regarded as nodes of the second order graphs, etc. Since parallel computers are considered, the highest order flow graphs describe the interconnections and information flow between (arrays of) processing elements, memories, and control units.

The work of a flow graph G (system or part of a system) is given by

$$\overline{w}(G) = \sum_{i=1}^{n} h_i \tilde{w}(\text{hyperedge } i) = \sum_{i=1}^{n} h_i \sum_{j=1}^{m_i} w(F_j)$$

where	n	is the number of hyperedges of G
	h_i	the label of hyperedge i ($\sum_{i=1}^{n} h_i=1$)
	m_i	the number of nodes of hyperedge i
	$w(F_j)$	the work of the j-th node (facility) of hyperedge i

Note that h_i may be for example the relative frequency of an instruction associated to hyperedge i, and that w is an arbitrary complexity measure.

The power $p(G)$ of G is the ratio of its work to the (average) cycle time of the corresponding system or part of the system.

Failures of components of a system normally mean the deletion of 'defective' hyperedges in the flow graphs involved. Then an investigation of the reduced flow graphs gives some information about the effect of the failures.

As an example, the work of an ILLIAC IV array is determined by means of the measure introduced by Hellerman [4]. Compare [1,3,9; 5,6,7,8,1o].

<u>Array with 64 PEs</u> 2o9o56 wits (for instructions with memory access; labels h_i=const.)

<u>Control Unit</u> 32o5 wits

Buffer 1315, Gating Block 65, ACARs 2o48, Address Adder 388, Shift/Logic Unit 1567, Decoder 32, Final Queue 512

<u>Processing Element</u> 2352 wits

Registers 3392, Address Adder 256, Artihmetic Unit 1792, Shift/Logic Unit 1567

<u>PE Memory</u> 821 wits

<u>Common Data Bus</u> 512 wits

<u>CU Bus</u> 4o96 wits

<u>Bus for Enable Signals</u> 16oo wits

References

[1] G. Barnes, et al., "The ILLIAC IV Computer" <u>IEEE Trans. Comp</u>. 17 (1968), pp. 746-757

[2] C. Bell, and A. Newell, <u>Computer Structures</u>, McGraw-Hill, (1971)

[3] R. Davies, "The ILLIAC IV Processing Element" <u>IEEE Trans. Comp</u>. 18 (1969), pp. 8oo-816

[4] L. Hellerman, "A Measure of Computational Work", <u>IEEE Trans. Comp</u>. 21 (1972), pp. 439-446

[5] L. Hellerman, "The Power and Efficiency of a Computer System", <u>GI-NTG Proceedings, Lecture Notes in Computer Science</u> 8, (1974), pp. 19o-2o5

[6] R. Hemmersbach, <u>Über ein informationstheoretisches Maß und seine Anwendbarkeit auf Rechnersysteme</u>, Univ. of Bonn, Diplomarbeit, (1975)

[7] R. Hemmersbach, and D. Schütt, "A Comparison of the Systems 36o-4o and 37o-168 by means of an Information Theoretical Measure", <u>Digital Processes</u> 1, (1975), pp. 329-332

[8] S. Hoener, "Zur Leistungsbewertung von Multiprozessor-Strukturen", <u>GI Proceedings, Lecture Notes in Computer Science</u> 26, (1975), pp. 396-4o5

[9] P. Mies, and D. Schütt, <u>Feldrechner</u>, B.I.-Wissenschaftsverlag, (to appear in 1976)

[1o] H.-H. Wielage, <u>Die Bewertung digitaler Systeme mit Hilfe der Rechenarbeit</u>, TU Munich, Diplomarbeit, (1976)

ERROR DETECTION AND RECOVERY IN A DATA-FLOW COMPUTER*

David P. Misunas
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- The highly modular structure of data-flow computer permits the inclusion of fault-tolerant capabilities at the module level within such a machine. Error detection and recovery is provided through redundant computation. The highly parallel structure of the routing networks, comprising the Memory/Functional Unit interconnection paths, allows reconfiguration of the processor upon detection of a faulty component with only slight degradation of either the performance or the ability to detect and recover from further errors. Due to the unique structure of the interconnection networks, the increase in size and complexity of a data-flow processor necessary to implement the fault-tolerant capabilities is less than the amount of redundant computation necessary to assure error recovery.

Introduction

Although the reliability of the components utilized in digital computers has increased tremendously in the past few years, in any system with such complexity, there is always a chance of failure. Also, in many applications it is either difficult to gain access to equipment for repairs or any failure would have catastrophic results, as in such machines as spacecraft computers and air traffic control computers. For these reasons, a great deal of effort has been devoted to fault-tolerant design techniques [1, 2, 3, 10, 11].

Most proposed fault-tolerant systems are composed of a number of processors executing different copies of the same program and comparing results. If a discrepancy arises, either a separate processor is utilized to check the system in an attempt to discover faulty components or the processors in the majority disable the minority processor(s) and continue with degraded fault recognition and recovery capability.

This approach to the structure of a fault-tolerant system has the problem that system reconfiguration is accomplished at a high level. The disabling of an entire processor upon detection of an error significantly degrades the fault recognition and recovery capabilities of the system unless there is a large number of redundant processors. However, the cost of maintaining many extra processors has thus far been rather prohibitive, allowing its introduction only in vital applications.

The data-flow processors described by Dennis and Misunas [5, 6] have a highly modular structure which permits system reconfiguration at a low level. Indeed, a completely fault-tolerant system can be implemented on a single processor through the use of program redundancy. System reconfiguration upon detection of an error is achieved by selecting alternate paths around failed components.

The occurrence of a failure in a data-flow processor and the associated bypassing of the faulty component do not affect the capability of the system to detect further errors and further reconfigure the processor in response to those errors as long as there remain enough fault-free components to perform the computation and there are enough paths connecting the memory and the functional units of the processor. Also, the additional cost of a computer incorporating such a solution over the cost of a non-fault-tolerant processor is much less than the additional cost of a system utilizing reconfiguration at the processor level.

Structure of the Data-Flow Processor

A data-flow processor is structured as a packet communication architecture [4]. Units of the processor communicate through the transmission of information packets, and delays in packet transmission do not affect the correct operation of the processor.

The data-flow program representation utilized as the base language of a data-flow processor allows the data-driven execution of a computation. An instruction of a data-flow program is enabled for execution only after the receipt of all required operands. Upon becoming enabled, an instruction proceeds to an appropriate functional unit which performs the desired computation and sends copies of the result to instructions which need them for execution.

In a companion paper [8], we examine the performance of an elementary data-flow processor designed to execute signal processing computations. More advanced data-flow processors, incorporating conditional and iterative constructs, data structures, and procedures are described in [6], [9], and [7], respectively. For the purpose of examining fault-tolerance techniques applicable to a data-flow computer, we will use the elementary processor shown in Figure 1.

*This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

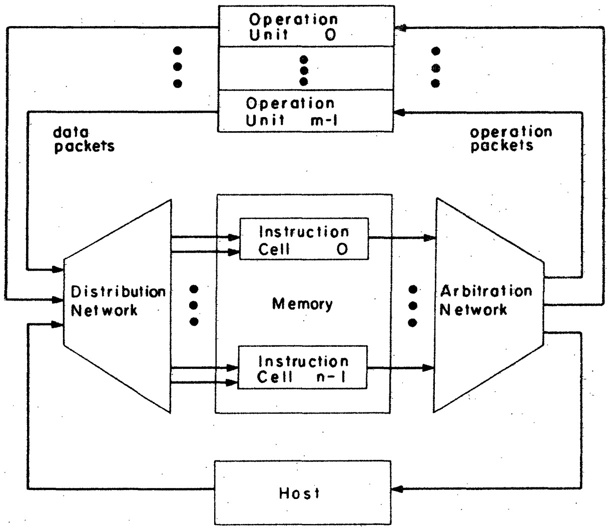


Figure 1. Structure of the elementary data-flow processor.

The Memory of the processor consists of a number of Instruction Cells, each composed of three registers (Figure 2) and holding one instruction of a data-flow program. The first register of an Instruction Cell contains the operation specification and the identifiers of destination Cells to which results of the operation are to be sent. The second and third registers contain space for the necessary operands and initially hold any values necessary to start the computation. An Arbitration Network conveys operation packets containing enabled instructions, each consisting of an operation specification, a number of destination addresses, and all required operands, from the Memory to the Functional Units. A Functional Unit performs the specified operation upon the operands in each operation packet received and forms a data packet, consisting of one copy of the result and a destination address, for each destination specified. The Distribution Network accepts data packets from the Functional Units and conveys them to the proper destination Instruction Cells.

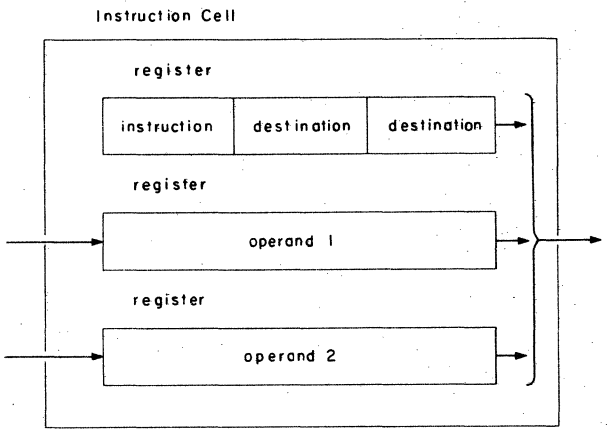


Figure 2. Structure of an Instruction Cell.

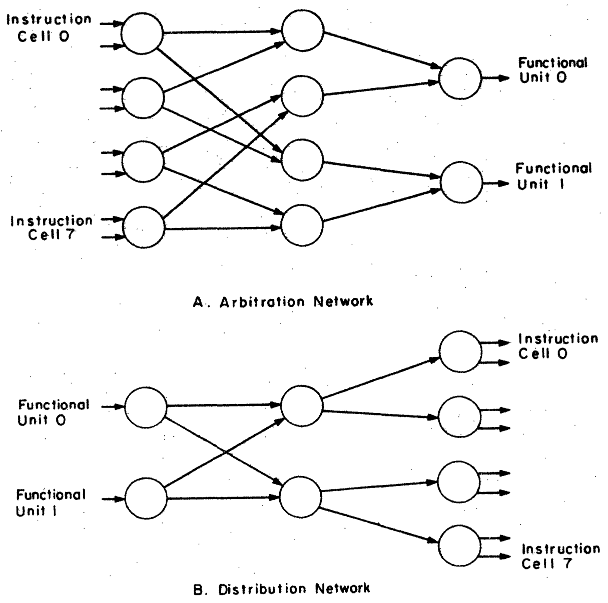


Figure 3. Structure of an elementary Arbitration and Distribution Network.

The Host computer is a conventional processor and is used to initialize the program in the data-flow processor, monitor its execution, and reconfigure the processor upon detection of an error. The Host performs these functions by means of packets sent through the networks. For the purposes of this discussion, we will assume that the Host is error-free. Such freedom from error in the Host can be achieved through more conventional techniques [10].

To examine the reconfiguration capability of a data-flow computer, we must understand the structure of the interconnection networks of the processor. An elementary structure of the Arbitration and Distribution Networks is presented in Figure 3, and the structure of a typical network node is shown in Figure 4. The arbitration unit in the node passes packets arriving at its input ports one-at-a-time to its output port, using a round-robin discipline to resolve any conflicts. Once a packet has been accepted by the arbitration unit of a node, it is stored in a buffer unit until the succeeding unit is ready to receive it. A switch unit directs a packet to one of the several possible next nodes, controlled by some property of the packet. In the Arbitration Network, the operation specification controls the switching, whereas in the Distribution Network, the switching is specified in the destination address.

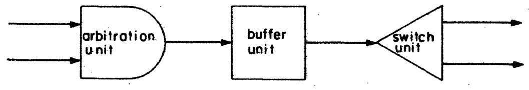


Figure 4. Structure of a network node.

The networks operate asynchronously, following a "hand-shaking" communication protocol for the transmission of packets. Once a packet has been accepted by a node and transferred to its buffer unit, the appropriate next node in the path is notified that a packet is ready to be transferred to it. No further action is taken until an acknowledge is returned from the succeeding node, at which time the entire packet is transmitted to that node.

Error Detection

There are three problems which must be faced in the introduction of fault-tolerant capabilities to a computer system. First, it must be possible to detect the occurrence of each error, whether it is caused by a hardware failure or by some other system malfunction. Second, the computer must be able to continue the computation in which the error occurred to a successful finish. And third, if a given error is caused by a failure of the hardware, the bad component(s) must be isolated to prevent the occurrence of further errors.

In this section we examine the solution to the first two problems of error detection and revovery. The next section discusses the reconfiguration of the processor in response to a determination that a component has failed. First, however, we must introduce a few terms. An error within a processor is the generation of an incorrect result. The errors that interest us here are those caused by faults; that is, by hardware malfunctions. Such faults are generally classified as either transient, intermittent, or permanent, depending upon their rate of occurrence. For a thorough discussion of these distinctions, see [2, 10].

Because inter-unit communication in a data-flow processor is asynchronous, a permanent fault and many types of transient and intermittent faults will halt the flow of packets through a particular unit of the processor. Such is the response to the well-studied and common problems of "stuck-at" faults, shorts, broken IC's, etc.

On the other hand, an error need not have such a readily distinguishable effect. It may only cause an incorrect result to propogate within a computation through the generation of an error in a packet, the production of extra packets, or the misdirection of a packet. To detect and recover from such errors, we need to introduce some amount of redundancy; that is, we must execute several copies of the computation simultaneously and compare the results through some voting process.

The elementary data-flow processor is designed to perform stream-oriented computation. Hence, for this processor we need to use triple redundancy techniques to allow us to not only detect, but also recover gracefully from errors. Triple modular redundancy (TMR) is one of the most widely utilized methods of fault-tolerant design. Generally, a system constructed in such a manner consists of triplicated hardware and triplicated voters. Each voter has three inputs, one from each of the three identical hardware modules. The implementation of these techniques within a conventional computer system has been widely described in the literature [2, 10], and we will not discuss it further here.

The application of TMR techniques to an elementary data-flow processor requires the triplication of several parts of the processor. There must be at least three functional units of each type, and a different functional unit must be utilized by each of the separate copies of a given program. Also, the size of the Memory must be tripled to accommodate the additional programs, and the programs must be distributed among the Instruction Cells of the Memory so as to insure that corresponding instructions of the various copies of a program do not share final stages of the Distribution Network or initial stages of the Arbitration Network.

The process of result comparison or "voting" is carried out at the Instruction Cells of the processor. In a data-flow processor with triple redundancy at the instruction level, each Cell receives three values destined for each operand register, one from each of the three copies of the previous instruction. The Cell incorporates a mechanism to compare the three values received and signal that an error has occurred if there is any discrepancy.

Upon detection of such an error by a Cell, if two of the values received agree, that value is used as the operand value, and an error message is sent to the Host. If all three operands are different, not only is an error message sent to the Host, but an error indicator is used in the operand field to indicate that an error has occurred, and the result produced by any Functional Unit or Memory Cell processing the error indicator is another error indicator, causing the error indicator to propogate through the remainder of the computation. Hence, the use of a TMR scheme does not permit recovery from two errors which simultaneously affect the same value in two of the three copies of a computation. However, the probability of such multiple errors is rather low.

An error message is sent to the Host in an error packet which travels through the Arbitration Network to the Host output ports of the network. The format of an error packet is as follows:

dest id = Host
unit id at which error occurred
error code

The Host analyzes the location and type of each error as described in the error packets to determ_ne the module at fault.

The level at which the outputs of redundant computations are compared largely determines the ease with which the Host can isolate the occurrence of a persistent error to a faulty component. If such checking is carried out at the instruction level, then the Host knows from the unit identifier at which the error occurred the exact path followed by the faulty result. However, to achieve this level of error detection requires increasing the number of packets flowing through the processor by a factor of nine due to the fact that each of the three identical computations transmits one copy of each result not only to the succeeding instruction(s) within that program, but also to the corresponding next instruction(s) of each of the two identical programs.

Comparing results at the end of a computation does not provide as much useful information about the location of an error; however, it only triples the number of packets flowing through the processor, and, as we shall describe later, the Host has other means at its disposal for finding faulty components.

Processor Reconfiguration

The addition of redundant connections within the network structures of a data-flow processor allows the routing of packets around network nodes which have failed. In Figure 5 we show one such structure for the Arbitration Network of Figure 2. The failure of a node is detected by the preceding node through examination of its buffer size. Since the arbitration units within a network node operate with a round-robin discipline, an upper bound on the time necessary to service a transmission request is obtained by multiplying the number of inputs to an arbitration unit by the packet transmission time. Within this time, only a small number N of packets destined for the node can be received by the preceding node. Thus, if either more than N packets destined to the succeeding node are received after sending a transmission request and before receiving an acknowledge or an amount of time greater than the arbitration unit processing time has passed (determined through the use of the check packets described in the next section), it can be assumed that the succeeding node has problems. Once this determination has been made, all packets destined for that node are rerouted around it, and the Host is notified through the transmission of an error packet.

In illustration, suppose node B of the network shown in Figure 5 fails. Either the buffer in node A will back up, or an amount of time greater than the packet processing time of node B will pass. Node A then sends all packets which it contains and which are destined for B to C. Node C sends the packets to the correct destination node, for example node D.

Node A also sends an error packet containing the network level number and the destination identifier of each rerouted packet to the Host, indicating that the succeeding node has failed somehow. Note that multiple failures cause the generation of multiple error packets.

The Host, upon determination that a node has failed, sends a <u>command packet</u> to the preceding node(s), ordering the node(s) to permanently bypass the failed node, and signals the user that a failure has occurred. This permanent reconfiguration of the processor halts the flow of error packets which have been generated each time the failed node has been bypassed. All packets contained in the failed node are lost; however, the redundant computation should produce the desired result.

More significant problems arise in the case of multiple failures of adjacent nodes of a network. In such a case, it may not be possible to reroute packets at the preceding node since there may be no nodes left which can provide an alternate path. Thus, rerouting must be ordered by the Host at an earlier level in the network and many more packets are lost.

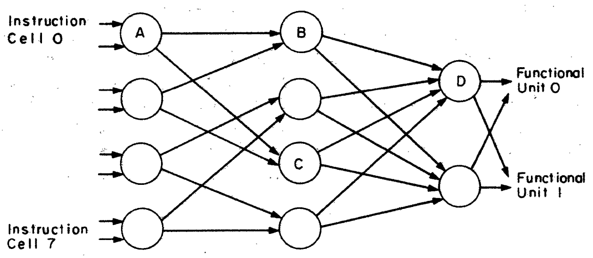


Figure 5. Structure of a fault-tolerant Arbitration Network

Thus far, we have only considered the failure of a portion of an interconnection network of a data-flow processor. For completeness, we must now consider the failure of a Memory Cell or a Functional Unit. Such a failure requires more intelligence on the part of the Host in order to correctly recover.

Failure of an Instruction Cell is detected by the Host through analysis of the messages contained in the error packets it receives. If a Cell fails completely, no destination will receive a result, and the Host, after checking the path which should have been followed by the operation and data packets in the manner described in the following section, can conclude that a problem exists within the Cell. Other failure modes of a Cell are determined through more extensive analysis of the error messages.

Failure of a Cell cannot be solved by the mere rerouting of packets destined for the Cell. The Host, upon recognition of the fact that a Cell has failed, must reconfigure the processor. The instruction contained in the Cell must be located in another Cell, and the destination addresses of the preceding instructions must be appropriately changed. Since the Host performed the initialization of the processor and has full knowledge of the memory status and content, this reconfiguration can be readily accomplished.

Complete failure of a Functional Unit also requires reconfiguration of the processor to bypass the bad unit. This reconfiguration is accomplished by resetting the switch unit feeding the Functional Unit until a new unit can be installed.

The use of pipelined Functional Units allows partial failure to be solved through reconfiguration of the Functional Unit itself. If a Functional Unit is constructed of stages which are capable of performing the same or similar functions (perhaps microprogrammed for a specific function) and the Unit contains a number of redundant stages, failure of a stage can be corrected by bypassing the stage. The necessary reconfiguration is accomplished by sending a command packet to the stage preceding the one which has failed. If it is not known which stage has failed, the unit can be reconfigured one stage at a time until it either operates properly or is determined to be unsalvagable.

Processor Verification

Through the Host's connection to the Arbitration and Distribution Networks, the integrity of the networks and the Functional Units can be assured. A check packet sent by the Host into the Distribution Network consists of two destination specifications. The first designates a path through the Distribution Network to an Instruction Cell. Upon reaching a Cell, a check packet is placed directly upon the Cell's output link and enters the Arbitration Network, where the remaining second destination address is used to direct the packet to an output port of the network connected to the Host.

To examine the operation of the Functional Units of the processor the Host maintains connections to the Arbitration Network inputs and the Distribution Network outputs. Operation packets are directed from the Host to specific Functional Units. The destination address contained in such a packet designates one of the output ports of the Distribution Network leading to the Host. In this manner, the Host can see if each Functional Unit is operating properly.

The periodic use of check packets not only supplies useful information to the Host as to the presence of permanent faults, but also provides a solution to one of the basic problems in fault-tolerant asynchronous design; that is, the difficulty of knowing how long to wait for a result in a redundant asynchronous computation. If a computation is being performed in a TMR fashion and two of the final results arrive, it is difficult to tell whether the third has been lost somewhere or is merely delayed. To determine this, we must somehow introduce the concept of time.

The structure of a data-flow processor is such that there is a readily determined upper bound on the time necessary for the execution of an instruction within the processor [8]. Thus,once we know the depth of a data-flow program; that is, the maximum length path from the first instruction to the last instruction, we can determine a maximum execution time for the computation.

Check packets sent by the Host through the Instruction Cells should have a time interval between successive packets destined to one Cell which is greater than the maximum instruction execution time. Then a Cell, merely by knowing the length of the maximum path between itself and the last Cell which performed a comparison, can readily determine if a packet has been lost. This determination is achieved by waiting an amount of time equal to the maximum instruction execution time multiplied by the path length after the receipt of the first operand. If the time expires and the other operands have not been received, the Cell becomes enabled without them, and an error packet is sent to the Host.

The Cost of Fault-Tolerance

To examine the penalty invoked in the implementation of fault-tolerant capabilities within a data-flow computer, let us consider the fault-tolerant structure of the simple processor presented in the companion paper on performance [8]. The elementary processor described in that paper contains 128 Instruction Cells, three level Arbitration and Distribution Networks, nine Functional Units, and can support a processing rate of approximately 28 MIPS.

The addition of fault-tolerant capabilities to such a processor structure requires increasing the number of Instruction Cells by a factor of three, to 384. Also, we are now required to have 27 Functional Units to maintain the throughput. The Arbitration and Distribution Networks must be increased in size to support both the additional memory and the larger number of packets flowing through the machine. However, we can support the additional Memory Cells with only a restructuring of the networks in this case, no additional stages are required.

The cost of an elementary data-flow processor is fairly evenly distributed among the four parts of the processor: the Memory, the Arbitration Network, the Distribution Network, and the Functional Units. The addition of fault-tolerant capabilities essentially triples the cost of the Memory and the Functional Units. However, the restructuring of the networks and the additional complexity within each node only increases the network cost by approximately 75%. Let c be the cost of the elementary data-flow processor without fault-tolerant capabilities, then the cost of a fault-tolerant version is:

$$2[3(.25c)] + 2[1.75(.25c)]$$
$$=2.38c$$

Hence, to introduce fault tolerance we have tripled the amount of computation performed with only a 140% increase in cost.

In a more advanced data-flow processor, incorporating procedure activation capabilities, fault tolerance can be implemented through dual redundancy. This is possible due to the ability to relocate a computation within the processor and restart it upon detection of an error. Hence, the cost of fault tolerance within such a machine includes doubling the Memory and Functional Units and increasing network cost by one-quarter to one-third, yielding a total cost increase between 5/8 and 2/3. With proper attention to detail, it may even be possible to obtain full fault tolerance with the long-sought-after 50% increase in complexity.

Concluding Remarks

The extension of the data-flow architecture to incorporate the ability to detect and recover from errors in a computation appears to be quite feasible. Though the study of this topic is by no means complete, preliminary results indicate that the cost of such an extension is very attractive, and the low-level reconfiguration utilized shows much promise in its ability to recover from a number of hardware failures before incurring significant performance degradation. The fault-tolerance techniques described herein are also applicable to other types of packet communication systems, such as the Packet Memory Systems described by Dennis [4], and the study of their incorporation is currently in progress.

Acknowledgement

The author wishes to express his appreciation to Ken Weng for many stimulating discussions and for the helpful comments on this paper.

References

[1] "An Application-Oriented Multiprocessing System," IBM Systems Journal (No. 2, 1967).

[2] Avizienis, A. "Design of Fault-Tolerant Computers," Fall Joint Computer Conference 31, AFIPS, New York, 1967, pp. 733-743.

[3] Avizienis, A., et al, "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Transactions on Computers C-20, (November, 1971), pp. 1312-1321.

[4] Dennis, J. B., "Packet Communication Architecture," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, (August, 1975), pp. 224-229.

[5] Dennis, J. B., and D. P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," Proceedings of the ACM 1974 National Conference, ACM, New York, (November, 1974), pp. 402-409.

[6] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York (January, 1975), pp. 126-132.

[7] Misunas, D. P., A Computer Architecture for Data-Flow Computation, SM Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., (June, 1975).

[8] Misunas, D. P., "Performance Analysis of a Data-Flow Processor," Proceedings of the 1976 International Conference on Parallel Processing, IEEE, New York, (August 1976).

[9] Misunas, D. P., "Structure Processing in a Data-Flow Computer," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, (August, 1975), pp. 230-234.

[10] Newmann, et al., A Study of Fault-Tolerant Computing: Final Report, SRI Project 1693, Stanford Research Institute, Menlo Park, California, (July, 1973).

[11] Wakerly, J. F., Low-Cost Error Detection Techniques for Small Computers, Technical Report No. 51, Digital Systems Laboratory, Stanford University, Stanford, California, (December, 1973).

RELIABILITY ASPECTS OF THE ILLIAC IV COMPUTER*

Iftikhar A. Baqai
and
Tomas Lang
Computer Science Department
University of California
Los Angeles, California 90024

Abstract -- The ILLIAC IV is an array computer used for Single Instruction Stream-Multiple Data Stream type computations. The large number of processing elements (PE's) in the system gives rise to a high probability of system failure. The use of dynamic redundancy techniques to make the system fault-tolerant is proposed. After a permanent fault is confirmed in a PE, the PE is removed and a spare is introduced into the system. Due to the highly structured interconnections between the processors, the replacement has to be done in a way that preserves this structure.

Two different replacement schemes describing the system recovery after a PE fails, are presented. One of the schemes is very general and can be applied to systems with interconnection networks different from that of ILLIAC IV. The circuit implementations of the recovery mechanism are also discussed. It is noticed that at the expense of a small amount of additional hardware, a considerable increase in reliability is obtained.

I. Introduction

ILLIAC IV is a parallel array computer containing four subarrays, each of 64 processing elements abbreviated as PE's. By using parallelism of operation, very high speeds in computing have been achieved [1].

The special feature of the ILLIAC IV is a common control unit which decodes the instructions and generates control signals for all the processing elements in the array. This eliminates the cost and complexity for decoding and timing circuits in each PE. Thus in ILLIAC IV, processing of various data streams is controlled by a single instruction stream. The need to exclude some data or process it differently is handled by providing each processor with an ENABLE flip-flop that controls the instruction execution at the processor level [1]. The processors in an array are labeled from 0 to 63. To facilitate data exchange, PE i has connections to four other processors $(i \pm 1)$mod 64 and $(i \pm 8)$mod 64. The interprocessor data transmission of arbitrary distance is accomplished by a sequence of routings specified in a single instruction.

We study the reliability aspects of the ILLIAC IV computer with respect to PE failures. In the present structure, a single PE failure causes a system failure and the large number of PE's (64) in an array gives rise to a high probability of failure. The system availability as well as its reliability can be enhanced by introducing hardware redundancy and thus making the system fault-tolerant. The application of static redundancy is ruled out because of economic considerations [2]. In the dynamic redundancy technique that we adopt, fault-caused errors are allowed to manifest themselves in the system. Fault-tolerance is then implemented by two successive actions. First, the presence of a fault is detected and then a recovery action takes place. If a restart of the program (rollback) fails to correct the error, a permanent fault is assumed and the faulty PE is removed [2, 3]. In this work, we limit our attention to the method of introducing spare PE's into the system to balance the removal of the failed PE.

We propose the use of one spare PE to tolerate all single PE failures. When a PE fails and it is removed from the system, the interconnection structure is perturbed. Assuming PE labeled x fails, then the routing interconnections of x to (x+1)mod 64 (from now on, all numbers of the type x+k are assumed to be modulo N when referred to a system containing N PE's), x-1, x+8 and x-8 are disturbed. So, for x+1, the data for routing by distance $\underline{+1}$ does not come from x and for x-8, the data for routing by $\underline{-8}$ does not come from x. Similarly for x-1 and x+8, the data for routing by distances $\underline{-1}$ and $\underline{+8}$ respectively does not come from x any more. We investigate three schemes by which the data routing ability is restored and, as a result, the ILLIAC IV system tolerates PE failures. The schemes are

i) Reorganization of the Interconnection Network (ROIN)
ii) Decoupling Network with Direct Replacement (DNDR)
iii) Decoupling Network with Indirect Replacement (DNIR)

In the ROIN scheme, after a spare becomes part of the system, the interconnection structure is reorganized so that the data routing proceeds unhindered as in the pre-failure situation. The disturbed interconnection structure is restored by providing extra interconnections and a selector circuit for all the PE's. The extra interconnections and the selector circuit form the redundant features of this scheme. In this scheme, the spare replaces the failed PE indirectly in the sense that it (spare) does not (generally) assume the label of the failed PE.

*This work was supported by the National Science Foundation under Grant No. MCS72-03633 A04.

In the Decoupling Network Schemes the data is routed to the correct PE by using an additional network which isolates the PE's from the interconnection network. In the DNDR case, the spare replaces the failed PE directly. The interconnections between the spare and other PE's are provided through a bus structure. In the DNIR case, the failed PE is replaced by the spare indirectly and in the same manner as in the ROIN Scheme. Here we use two selector circuits for a PE. The decoupling network forms the redundant feature of the DN Schemes.

The ROIN Scheme involves modification of the interconnection network and is therefore dependent on the specific interconnection network of the system. On the other hand, the DN Schemes are very general and can be easily adapted to any system.

The ILLIAC IV interconnection network and its fault-tolerant versions are described in Section 2. The extension of the above schemes to interconnection networks different from that of ILLIAC IV is considered in Section 3. Some reliability estimates are given in Section 4 which indicate substantial improvement in the system reliability due to the introduction of redundancy. It is observed that the additional hardware required is very simple, so the increased system cost and complexity is minimal. The schemes are compared with respect to the complexity of their implementation in Section 5. It is concluded that the DN Schemes are easier to implement than the ROIN Scheme. The main draw-back with the DNDR Scheme is that it requires a duplex bus structure which has connections to all the PE's in the system. This fact has critical implications in the system design because of circuit limitations on the number of connections a bus may have. Additionally, the length of the bus line for such a system may require the use of repeaters to overcome the problems of noise contamination and attenuation of the signal values. The length of the bus line is also important in determining the signal propagation delays and may affect the speed of the whole system. The disadvantage of the DNIR Scheme is that the two selector circuits associated with each PE introduce logic delays which may slow down the system. The ROIN Scheme is the faster of the three schemes as it involves modifying the interconnection network and as such does not affect the speed of the system. The disadvantage of this scheme is that the number of interconnections required for a PE is twice the number required for the non-redundant ILLIAC IV. This feature adds to the system complexity.

The comparison of the schemes indicate that the DN Schemes are superior if the associated circuit delays are acceptable. For a fast fault-tolerant system, however, the ROIN Scheme might be more practical.

2. Restructuring of ILLIAC IV for Tolerance of a Failed PE

In this section, a structure of 64 PE's is described where PE i is connected to PE's $i\pm1$ and $i\pm8$. Three schemes are presented aimed at making the system tolerate failed PE's. The schemes are studied with respect to the steps involved in the recovery procedure and the complexity of its implementation.

There are sixty-four physical locations numbered from 0 to 63. The PE in the ith location has 'i' as its physical label while L(i) denotes its logical label. For the non-redundant system, L(i) = i for all i. The logical relabeling is done when a spare becomes part of the system. Unless otherwise specified, the word label refers to the physical label. The data for i for routing by distances -8, -1, +1 and +8 comes from PE's (i+8), (i+1), (i-1) and (i-8), respectively.

The above routing is disturbed if any PE fails, causing a system breakdown. To make the system tolerate PE failures, we introduce redundancy by adding spares to the original system. Two different strategies are used to restore the system back to its original configuration. We consider the case of single PE failures only so that one spare is sufficient. In both these situations as PE x fails, the system enters the recovery mode. If a restart of the program (roll-back) fails to correct the error, a permanent fault is assumed, the failed PE is removed and the spare becomes part of the system. The spare is originally labeled 64.

The restructuring of the system consists of three principal steps:

i) Relabeling strategy
ii) Transmission of the failure information
iii) Reconfiguration scheme

When a PE fails and a spare is introduced, relabeling is performed by assigning a new label to the PE's. It can be done in many ways but we discuss only two. In the Direct relabeling strategy, the spare assumes the logical label of the failed PE while all other PE's retain their original labels. In the Indirect relabeling strategy, a PE whose label is less than the failed PE's label retain its original label, while the others have their labels decreased by one. In this strategy, the spare is relabeled 63 (Fig. 2.1).

The information about the location of the failed PE can be transmitted to other PE's either by the failed PE itself (Local Transmission) or by the Central Control Unit.

We discuss two methods by which the system reconfigures itself after a spare becomes part of the system. The ROIN (Reorganization of the Interconnection Network) Scheme involves modifying the interconnection network to preserve the data routing ability. In Fig. 2.2(a), a schematic representation of the non-redundant ILLIAC IV is shown while in Fig. 2.2(b) the redundant system using the ROIN Scheme is presented. The other schemes, the Decoupling Network Schemes, are based on separating the interconnection network from the PE's. The interconnection network has 64 ports, one corresponding to each PE. The decoupling network is used to maintain, after relabeling, the correspondence between the PE's and the ports. The redundant ILLIAC IV model using decoupling network is shown in Fig. 2.2(c).

In designing a system which tolerates PE failures, a strategy has to be chosen for each step. In general, all choices described are feasible. Based on our analysis, we selected the ones leading to simpler and more elegant solutions. All schemes considered use local transmission of failure information. One of the schemes uses indirect relabeling and reorganization of the interconnection network. The second employs direct labeling and decoupling network, and the third, indirect relabeling and decoupling network. The schemes are now described and they are compared in Section 5.

2.1 Reorganization of the Interconnection Network (ROIN)

In this scheme the spare is connected serially with the active ones at one extreme end of the system as shown in Fig. 2.1. Whenever a failure occurs, the failed PE is eliminated and indirect relabeling is done. Extra interconnections are provided to the PE's so that after relabeling, the interconnection network is restructured to its original data routing ability. Each PE has a selector circuit built into it which chooses the correct data input for the respective PE out of all the inputs it receives. A selector circuit block diagram is shown in Fig. 2.3. The failure information is transmitted in the local mode.

Extra Interconnections. Fig. 2.1 shows that some PE's which were originally at distance eight with respect to each other, after relabeling are at distance seven only. Similarly, some PE's previously at distance nine are at distance eight now. This suggests the need for extra interconnections for all such PE's so as to permit data routing by distances ±8 after relabeling has taken place. Similar interconnections are also required for distance ±1 routing.

In addition to the four interconnections for every PE for the non-redundant ILLIAC IV, the fault-tolerant version has four extra interconnections. Some of these extra interconnections are used in the routing of the data whenever a spare becomes part of the system. The extra interconnections are listed in Table 2.1.

Reconfiguration Algorithm. A selector circuit for a PE receives data inputs from eight other PE's and selects one of these as the correct data input.

The selection of the correct input and hence the implementation of the selector circuit is based on the Reconfiguration Algorithm which defines the modified control for routing by distances ± 1 and ± 8. The disturbed interconnections for each routing are described separately in Table 2.2. In the table, x stands for the failed PE's label while the entries in the column i stand for the affected PE's. The column 'Source' tells about the new source of data. It is implied that for all PE's not mentioned in the table, the data, for routing by distance k (where k = ± 1 and ± 8), still comes from i-k.

Selector Circuit. The Selector Circuit receives the information about the failed PE as transmitted by it to the affected PE's. Then using the routing control signals, it selects one of the inputs as the correct input for the PE.

As indicated in the Reconfiguration Algorithm (Fig. 2.1), when x fails, the routing interconnections for PE's (x-1) to (x-8) and (x+1) to (x+8) are affected. In addition, the routing interconnections for PE's 0 to 7 and 56 to 63 are also affected. The above observation holds true for $8 \le x \le 55$. Similar results are obtained when $0 \le x \le 7$ and $56 \le x \le 63$.

To restore the ±1 routing, a failed PE transmits signals of the type $(i \pm 1)_f = 1$ to the affected PE's. If PE i receives $(i \pm 1)_f = 1$ it means that the PE $(i \pm 1)$ has failed. For the restoration of ± 8 routing, a more elaborate arrangement is required. To communicate with the affected PE's, a failed PE issues all or some of the six signals as shown in Fig. 2.4. The 'ℓ' signal is transmitted to all the PE's to the left of x (in the same row). The 'r' signal is transmitted to all the PE's to the right of x (in the same row). The 'u' signal is transmitted to the PE above x in the immediate upper row and to all the PE's to its right. The 'd' signal is, similarly, sent to the PE below x in the immediate lower row and to PE's to its left. The signals f_t and f_b are sent to all the PE's in the top row and the bottom row whenever x is in rows one to six.

The above scheme is chosen in such a way that it isolates the affected PE's. The changes in the data routing control for these PE's can be implemented simply on the basis of the received signals. Thus the modified routing for the PE's in the top row, middle rows and the bottom row becomes a function of the received signal values as listed in Table 2.3. In case a PE does not receive a combination of the specified signals, it simply means that the given PE has not been affected by the failure.

The Selector Circuit based on the above specifications can be designed and the number of gates used for an M-bit word is approximately 8M when wired-OR technology is assumed.

If x is in the top or bottom row, f_t and f_b signals are not transmitted. Also if x is in the top row, u is received by the bottom row PE's as shown by a dotted line (Fig. 2.4). Similarly if x is in the bottom row the d signal is received by the PE in the top row as indicated by a dotted line in Fig. 2.4.

2.2 Decoupling Network Approach

As indicated in the beginning of the Section, the ILLIAC IV can be characterized as a system with two blocks (Fig. 2.2(a)) where PE's either transmit data to or receive data from the interconnection network. In the redundant version of the system, the two blocks are separated by the decoupling network (Fig. 2.2(b)). The relabeling can be either Direct (DNDR) or Indirect (DNIR). The two strategies are discussed separately. In the Decoupling Network implementation, the interconnection network is regarded as consisting of 64 ports labeled in a one-to-one correspondence with the PE's. Each port has an entry point (data reception) and an exit point (data transmission) as shown in Fig. 2.5(a). Thus these schemes involve maintaining the one-to-one correspondence between the PE's, after relabeling, and the interconnection network ports.

2.2.1 Decoupling Network Indirect Replacement Scheme (DNIR)

As shown in Fig. 2.5(a), the data is transmitted to and from a PE to the corresponding port in the interconnection network. When the PE X fails, the indirect relabeling is performed accompanied by some switching to maintain the connections between the relabeled PE's and the corresponding ports. The new data routing is shown in Fig. 2.5(b). The switching action is accomplished by using two selector circuits for each PE-Port pair; one selector circuit is placed at the PE end, the other one at the port end. The information about the failed PE is transmitted locally. A signal is sent to PE's with labels greater than the failed PE. The selector circuits use this signal to effect the switching.

2.2.2 Decoupling Network Direct Replacement Scheme (DNDR)

In the DNDR Scheme, the failed PE is replaced by the spare directly and so it (spare) assumes the logical label of the failed PE. The connections between the PE's and ports when PE X fails are indicated in Fig. 2.5(c). In order to tolerate a failure in any PE, the spare should be able to connect to any port in the interconnection network. This function is performed by using a duplex bus structure which connects a spare to all the ports through some interfacing hardware. This hardware is rather simple and consists of a few gates. The information about the failed PE is transmitted locally to the corresponding port.

2.3 Comparison of the Schemes

The conclusion derived from the comparison of the schemes done in Section 5 is that the Decoupling Network Schemes are simpler to implement because these avoid the complexity of extra interconnections as required for the ROIN Scheme. For large values of N the DNIR Scheme seems more suitable because of fan-out and delay considerations associated with the DNDR Scheme. However, if the system requirements call for a design which does not compromise the speed of the system, then the ROIN Scheme is probably the best as it does not introduce any delays into the system operation. One need recall that the ROIN Scheme involves modifying the interconnection network, while for the DN Schemes, the selector circuit (DNDR) and the bus and other assorted gates (DNIR) cause additional logic delays and slow down the whole system.

3. General Schemes

Our discussion so far has been with reference to the interconnection network employed for ILLIAC IV only but the nature of the proposed schemes is such that they can be applied to other interconnection networks [6]. In general any PE i receives data from n PE's labeled j_1 through j_n. Similarly data is routed from PE i to n other PE's labeled k_1 through k_n.

In case of the ROIN Scheme, the PE 'i' in the fault-tolerant version of the system receives data from (or transmits data to) additional n_1 PE's. The number n_1 is a function of n (the number of interconnections for a given PE in the original system) and the type of interconnection network used. When a spare becomes part of the system and logical relabeling is done, the interconnection structure of the system is disturbed. Some or all of the extra n_1 interconnections provided to the PE's, are then used to reorganize the interconnection network. In the non-redundant system a PE can receive data directly from (or transmit to) n other PE's but in the redundant version, the PE receives data from (or transmits to) $n+n_1$ other PE's. The n_1 extra interconnections are not intended to add to its original routing ability but to preserve it in event of a failure. In this scheme a selector circuit (which is a part of the PE) receives data from $n+n_1$ PE's and depending upon the control signals and the failure information, chooses the correct input for the PE 'i'. The schematic diagram for this scheme is shown in Fig. 3.1.

In case of the Decoupling Network Scheme, the PE's and the interconnection network are separated by the decoupling network. Furthermore each port in the interconnection network corresponds to a given PE in the non-redundant system. The DN is designed such that it maintains one-to-one correspondence between the PE's and the ports. In the DNIR strategy, a PE can receive data from any of the two ports. The selector circuit at the receiving end of the PE selects the data from the correct port by using the information about the location of the failed PE. Similarly a port in the interconnection network can receive data from two PE's, and a selector circuit at that end performs the selection. In the DNDR strategy, the bus is connected to all the interconnection network ports and it can accept data from one PE at a given time to be passed on to the spare. Similarly the data may be transmitted by the bus (which gets it from the spare) to any one PE at a given time.

For the ROIN Scheme, by increasing the number of extra interconnections provided to the PE's and using suitable number of spares, theoretically any number of failed PE's can be tolerated [5]. It may be pointed out, however, that circuit limitations and complexity may become the limiting factor. In case of DNIR Scheme, for a system to tolerate m PE failures, the selector circuit at the receiving end of the PE would select data coming from one of the m ports of the interconnection network. Similar selection would be done at the entry end of the interconnection network ports. For the DNDR Scheme, the tolerance for any number of PE failures may be achieved by using multiple bus-spare pairs.

A significant advantage of the DN Schemes is their independence **from** the type of interconnection network used. Here the emphasis is on isolating the interconnection network and maintaining the one-to-one correspondence between the PE's and the ports of the interconnection network so that the decoupling network depends only on the relabeling. The disadvantage is the added circuit delays caused by the selector circuit and the bus. The ROIN Scheme is based on the modification of the interconnection network and hence depends on the specific interconnection network. This scheme is, thus, less adaptable and for each system a new design is required. The advantage of the ROIN Scheme lies in its being the fastest of the three schemes presented.

4. Reliability

In this section, we evaluate the improvement in reliability of the ILLIAC IV System, obtained for different values of the module (PE) reliability. As mentioned before, we only consider failures in the processing elements. The degradation in the overall system reliability due to the failure possibilities in the interconnection network, Central Control Unit etc., has not been considered. It is assumed that the introduction of fault-tolerant features into the PE hardware (and into the system) does not cause any change in the PE reliability; that unity coverage is provided and the failure rate of all the active and spare PE's is the same. The following notation is employed throughout the section.

$R_{PE}(t) = e^{-\lambda t}$ Reliability of a single PE, where λ is the failure rate. The time dependence is implicit even when 't' is not indicated.

$Rnr = R_{PE}^{64}$ Reliability of the non-redundant system.

$Rr = R^{64}(65-64R\ \ )$ Reliability of the redundant system that completely tolerates one PE failure.

$RIF(\lambda t) = \frac{1-Rnr}{1-Rr}$ Reliability Improvement Factor with respect to tolerance for one PE failure.

$MTI = \frac{Tr}{Tnr}$ where $R = Rnr(Tnr) = Rr(Tr)$. Mission Time Improvement when the system completely tolerates one PE failure.

The results are shown in Table 4.1. It is noticed that the RIF increases with the reliability of the non-redundant system. For $\lambda t = 10^{-4}$, the improvement in reliability as reflected by RIF, is better than two orders of magnitude. There is also significant improvement in the mission time as indicated by MTI.

The results indicate also that for a practical mission duration, the processing elements should have a rather low failure rate. For example, for the redundant system, a mission time of 10^2 hours at a reliability of .99 requires a module failure rate of 2×10^{-5} failures/hour. If this is not attainable, tolerance for more than one failure may have to be incorporated in the system.

The results shown in Table 4.1 indicate the improvement in reliability obtained when the system tolerates all single PE failures. The additional hardware (used in the redundant version of the system) consists of one PE and some selection circuitry in all the PE's and is insignificant. Thus appreciable improvement in system reliability can be achieved by implementing the suggested schemes in the ILLIAC IV System.

5. Conclusions

In this Section we compare the proposed schemes and then comment about the feasibility of their implementation.

(1) For complete tolerance for single PE failures, only one spare is required for all the schemes.

(2) For the ROIN Scheme, the PE's are provided with four extra interconnections along with a selector circuit. In the DNIR Scheme, a decoupling network consisting of two selector circuits for each PE is required. The DNDR Scheme uses a duplex bus structure and some interfacing hardware.

(3) The ROIN Scheme uses about 8M gates for an M-bit word for one selector circuit. The amount of hardware used for the DNIR Scheme is 4M gates while the DNDR Scheme requires a duplex bus structure and 3M gates.

(4) The asymmetries in the ROIN Scheme may necessitate changes in some of the inputs for certain PE's. This aspect may have implications in the mass production of the system. The DN Schemes, on the other hand, are symmetrical and the same kind of circuitry may be used for all PE's.

(5) The ROIN Scheme does not introduce any logic delays into the system speed as it is based on the modification of the system interconnection structure. The DNIR Scheme uses two selector circuits for every PE-interconnection-network-port pair and introduces two logic delays into the system. In case of the DNDR Scheme, all interconnection network ports have a data input/output connections to the bus which also acts as an 'OR' gate. Due to the circuit limitation, a bus can

have a restricted number of data connections, so as N increases the DNDR Scheme becomes less attractive. It may be interesting to note that the use of TRI-STATE logic [4] in the output stages allows at least 128 outputs to be connected to a single bus. Additionally a long bus is needed to connect all the PE's thus necessitating the use of repeaters. Here again, the QUAD-DRIVER of the TRI-STATE logic family can drive up to 1000 feet of bus line. Thus with the judicious choice of logic and other design parameters, the above problems can be overcome without having to add additional hardware. For large values of N, however, the length of the bus can significantly affect the speed of the operation of the whole system. In such situations the ROIN Scheme seems preferable.

(6) The DN Schemes are easily adaptable to other interconnection networks while a redesign of the system is required for the ROIN Scheme because of the rigid nature of the extra interconnections.

(7) The DN Schemes are easily extensible to the case of double (multiple) failures while the ex   is not that obvious in case of the ROIN Scheme.

In the light of the above comparison, our conclusion is that the DN Schemes are more general and easier to implement. The major advantage of the ROIN Scheme is its speed. It is noticed that in general, at the expense of small amount of additional hardware, tolerance for single PE failures is achieved. Tolerance for multiple PE failures is also attainable but more detailed and thorough investigation is required into the cost-effectiveness aspects of the problem to determine the optimum amount of tolerance for any given system.

Tables

PE's	Extra Interconnections to
$i = 0, 1$	$i+2$, $i+7$, $i+9$, 64
$2 \leq i \leq 6$	$i\pm2$, $i-7$, $i+9$
$i = 7, 8$	$i\pm2$, $i+9$, 64
$9 \leq i \leq 54$	$i\pm2$, $i\pm9$
$i = 55, 56$	$i\pm2$, $i-9$, 64
$57 \leq i \leq 61$	$i\pm2$, $i-9$, $i+7$
$i = 62, 63$	$i-2$, $i-9$, $i+7$, 64

Additionally PE 64 is connected to 0, 1, 7, 8, 55, 56, 62, 63.

Table 2.1 Extra Interconnections for the ROIN Scheme as Applied to the ILLIAC IV

+1 Routing

x	i	Source
0	1	64
1 to 62	0	64
	x+1	i-2
63	64	62

-1 Routing

x	i	Source
0	63	64
1 to 62	x-1	i+2
	63	64
63	62	64

+8 Routing

x	i	Source
0	8	64
1 to 7	0 to x-1	i-7
	8	64
	9 to x+8	i-9
8 to 56	0 to 6	i-7
	7	64
	x+1 to x+8	i-9
57 to 62	x-56 to 6	i-7
	7	64
	x+1 to 64	i-9
63	7	64
	64	i-9

-8 Routing

x	i	Source
0	56	64
1 to 7	0 to x-1	i+9
	56	64
	57 to x+56	i+7
8 to 56	x-8 to x-1	i+9
	56	64
	57 to 64	i+7
57 to 62	x-8 to 54	i+9
	55	i+9
	x+1 to 64	i+7
63	55	64
	64	i+7

Table 2.2 Reconfiguration Algorithm for the ROIN Scheme

+8 Routing

i	Received Signal	Source
0 to 7	$\ell + f_t + d$	i-7
8 to 55	d + r	i-9
56 to 63	d + r	i-9

-8 Routing

i	Received Signal	Source
0 to 7	$\ell + u$	i+9
8 to 55	$\ell + u$	i+9
56 to 63	$r + f_b + u$	i+7

Table 2.3 The Relationship Between the Affected PE's and the Received Signals for the ROIN Scheme

λt	R_{PE}	Rnr	Rr	RIF
10^{-4}	.99990	.99362	.99998	308
10^{-3}	.99900	.93800	.99800	31.1
10^{-2}	.99004	.52729	.86308	3.45
10^{-1}	.90484	1.6615×10^{-3}	1.1781×10^{-2}	1.01

(a)

R	λTr	λTnr	MTI
0.9	8.245×10^{-3}	1.646×10^{-3}	5
0.99	2.303×10^{-3}	1.560×10^{-4}	14.7

(b)

Table 4.1 (a) Reliability Improvement for the ILLIAC IV System

(b) Mission Time Improvement for the ILLIAC IV System

References

[1] G.H. Barnes, "The ILLIAC IV Computer," IEEE Transactions on Computers, Vol. C-17, August 1968, pp. 746-757.

[2] A. Avizienis, "Design of Fault-Tolerant Computers," AFIPS Conference Proceedings, Vol. 31, Thompson Books, Washington D.C., 1967, pp. 733-743.

[3] A. Avizienis, "Architecture of Fault-Tolerant Computing Systems," International Symposium on Fault-Tolerant Computing, Paris, 1975, pp. 3-16.

[4] "Characteristics and Applications of TRI-STATE IC's," National Semiconductor Corporation, 1972.

[5] I. Baqai, "Reliability and Length of Interconnection in ILLIAC IV Type Array Processors," Master's Thesis, Computer Science Department, UCLA, 1976.

[6] T. Lang, and H.S. Stone, "A Shuffle-Exchange Network with Simplified Control," IEEE Transactions on Computers, Vol. C-25, January 1976, pp. 55-65.

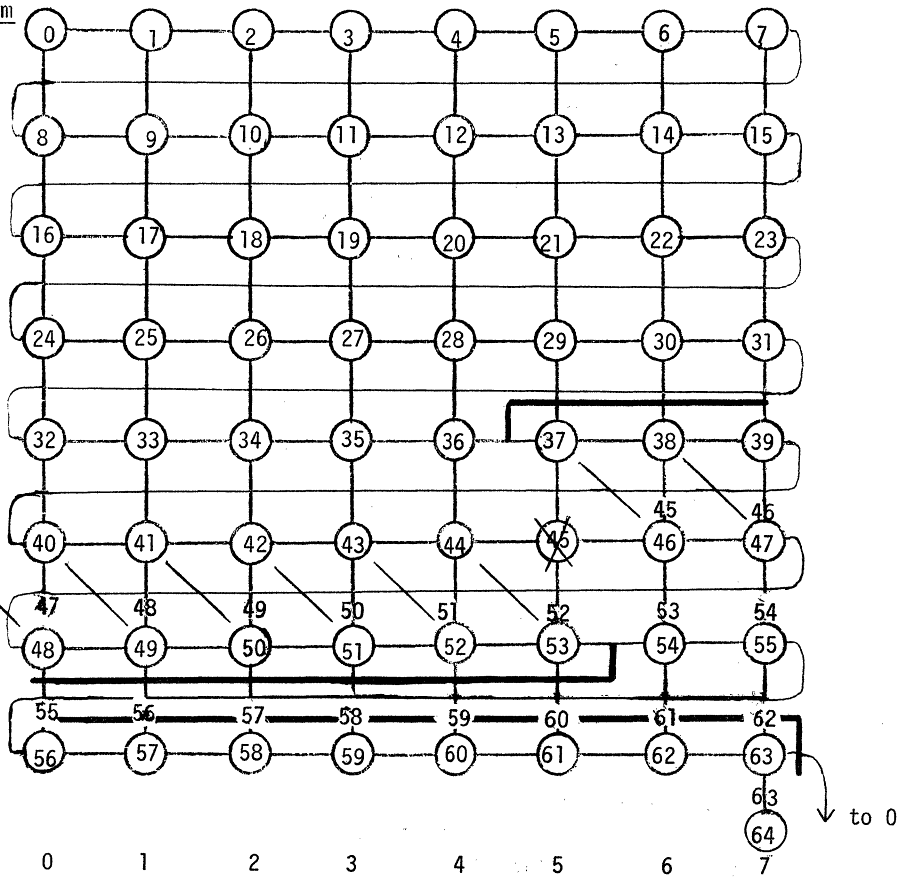


Figure 2.1 The PE's Affected by the Failure of X in ROIN Scheme (The PE's inside the parenthesis represent the affected PE's)

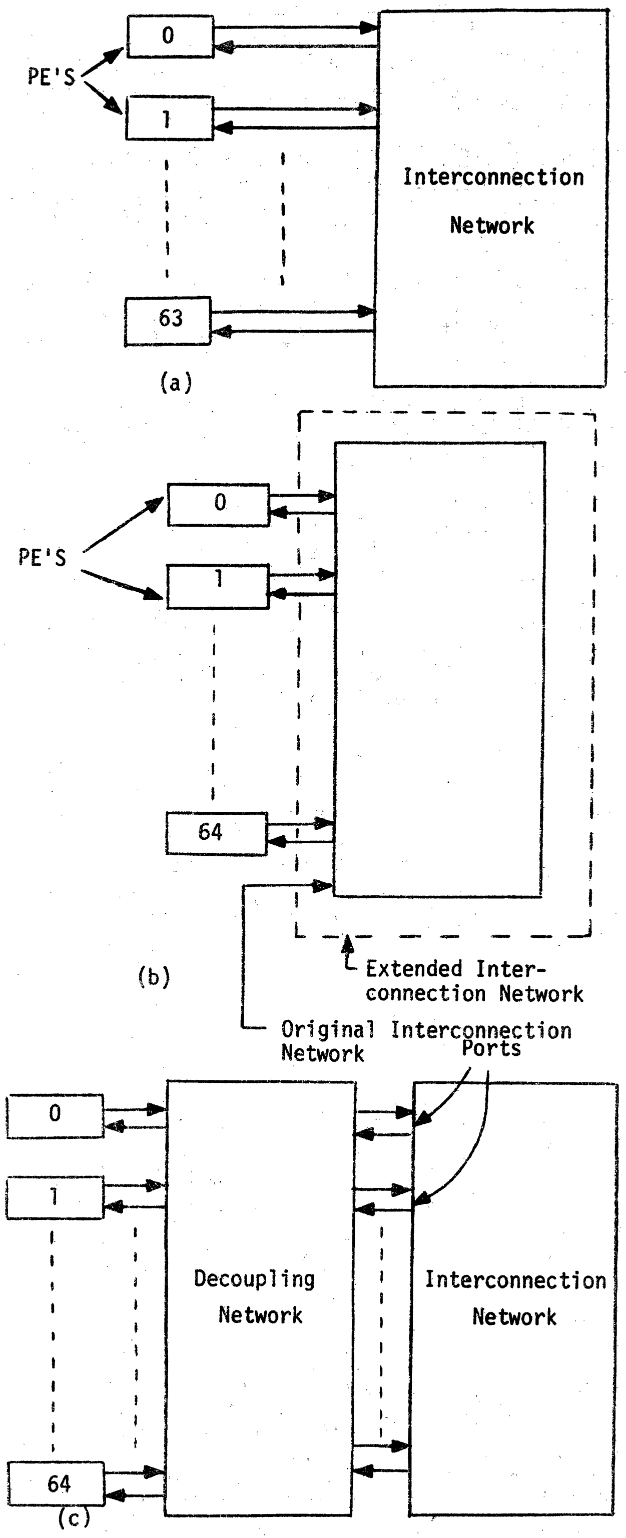


Figure 2.2 (a) Non-redundant ILLIAC IV System
(b) Redundant ILLIAC IV System Using the ROIN Reconfiguration Scheme
(c) Redundant ILLIAC IV System Using the DN Reconfiguration Scheme

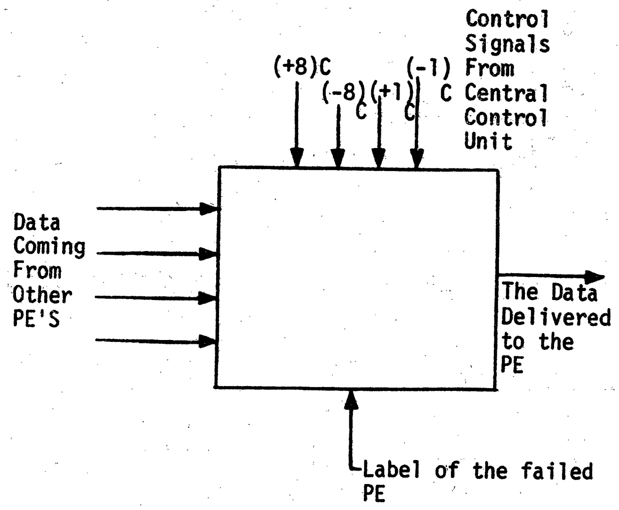

Control Signals From Central Control Unit
(+8)C
(-8) C
(+1) C
(-1) C
Data Coming From Other PE'S
The Data Delivered to the PE
Label of the failed PE

Figure 2.3 Selector Block Diagram for the ROIN Scheme

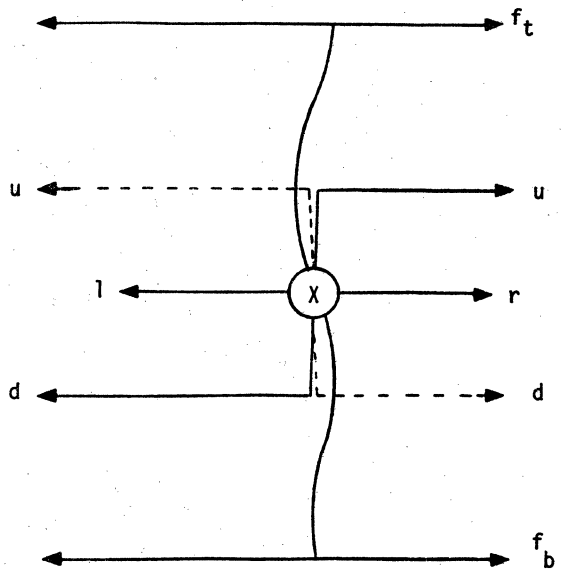
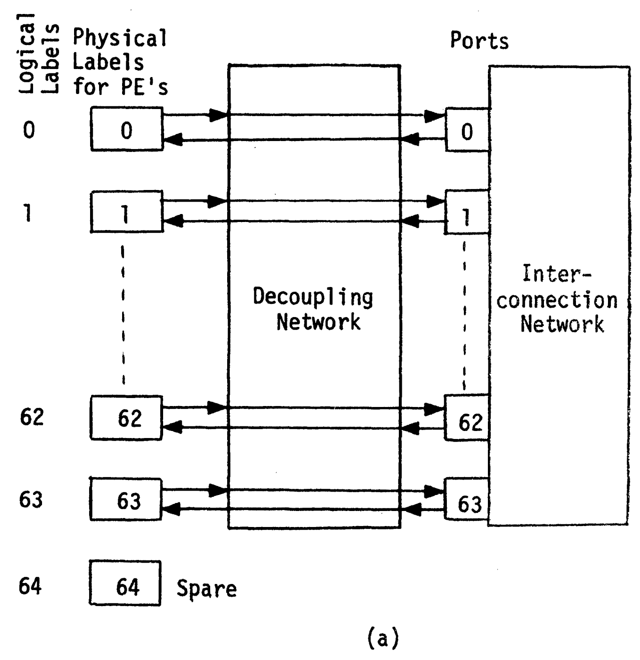


Figure 2.4 Transmission of the Signals by the Failed PE to the Affected PE's for the ROIN Scheme



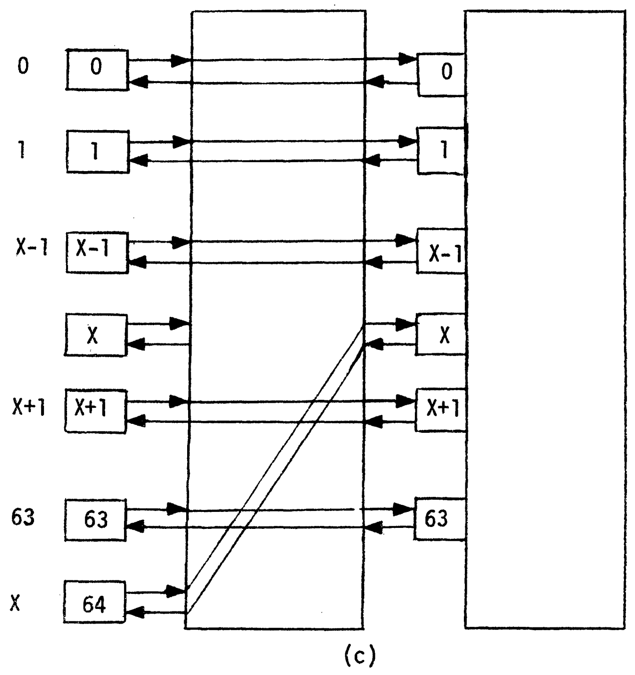
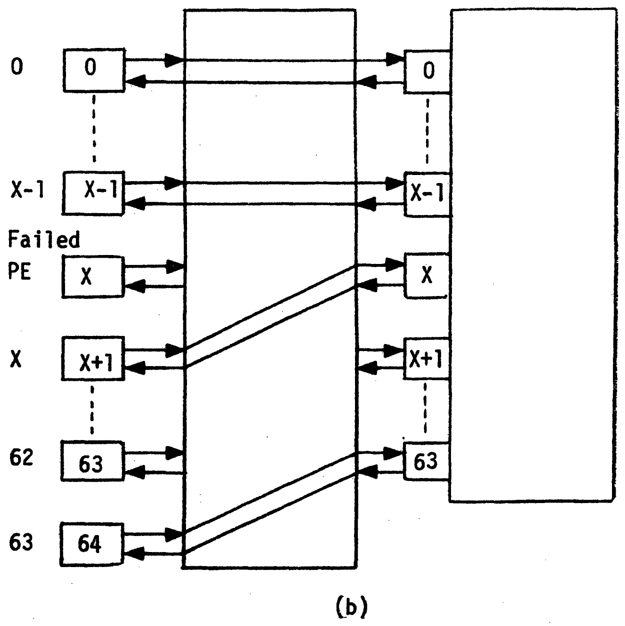


Figure 2.5 (a) The Role of a Decoupling Network in a Fault-Free Situation

(b) The Routing of the Data Through the Decoupling Network for DNIR Scheme

(c) The Replacement of the Failed PE in the DNDR Scheme



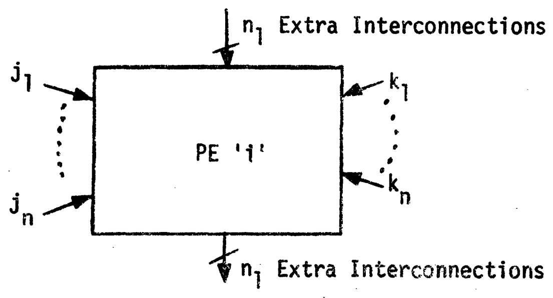


Figure 3.1 Schematic Representation of the General ROIN Scheme

COMPUTER ARCHITECTURES FOR ADVANCED AIR TRAFFIC CONTROL APPLICATIONS

Andres Zellweger
Advanced Concepts Staff
Federal Aviation Administration
Washington, D. C. 20591

Abstract -- This paper investigates multi-mini (or micro) processor configurations suitable for advanced highly reliable ATC applications. The stage is set by a characterization of the type of processing that must be performed and by a description of the design constraints imposed by the Federal Aviation Administration (FAA) operations and maintenance environment. Multiprocessor designs are developed against this background at the processor-memory level. Two specific examples, one a proposed design of an airborne computer for a future conflict avoidance system and the other a prototype surveillance site processor to be built by Texas Instruments as part of a recently awarded contract, will be alluded to throughout the discussion.

Introduction

Air Traffic Control (ATC) data processing requires highly reliable computers capable of both real-time radar data processing and lower priority processing of aircraft track and flight data. Output is normally directed to a dynamic graphic Plan View Display (PVD) or, in some advanced systems, to a small cockpit display for use by air traffic controllers and pilots respectively. Today this is done with suitably adapted early third generation medium-to-large scale general purpose multiprocessors.(1) Current technology trends towards fast microprocessors and low cost mini-processors offer a variety of new approaches to ATC data processing that promise higher reliability and greater economy in hardware, software, and maintenance cost.

This paper investigates multi-mini (or micro) processor configurations suitable for advanced, highly reliable ATC applications. The stage is set by a characterization of the type of processing that must be performed and by a description of the design constraints imposed by the Federal Aviation Administration (FAA) operations and maintenance environment. Multiprocessor design considerations are developed against this background at the processor-memory level. Two specific examples, one a proposed design of an airborne computer for a future conflict avoidance system and the other a prototype surveillance site processor to be built by Texas Instruments (TI) as part of a recently awarded contract, will be alluded to throughout the discussion.

Processing Characterization

The ATC computers addressed in this paper have three functions: control of a beacon interrogation system; processing of beacon radar data to track aircraft; and detection and resolution of conflicts (i.e., collision threats) between aircraft. In the case of airborne systems the computer drives a cockpit display to present relative position and altitude information about nearby aircraft and maneuver commands for the resolution of conflicts. In the case of a ground-based system similar information is transmitted to the aircraft via the beacon interrogation system for display in the cockpit. This form of ground-based, automated collision avoidance is known as Intermittent Positive Control (IPC). A discrete address beacon system (DABS) that permits message transmission to and from individually selected aircraft is assumed in a ground-based system. Detailed descriptions of the DABS, IPC, and airborne collision avoidance systems can be found in (2), (3) and (4). The current terminal (ARTS III) and enroute (NAS) ATC computers perform quite similar radar data processing functions. Conflict detection logic is operational in the NAS system and under development for ARTS III. The major difference is that in the current systems a controller is in the loop between the computer and the pilot. The primary output device of the NAS and ARTS III systems is a dynamic graphics display that shows the position, altitude, identification, and course of all controlled aircraft to the controller. He determines conflict resolution maneuvers and has the responsibility for commands, sent over VHF radio, to the pilot. Clearly, reliability requirements in the advanced system that automatically generate and transmit commands to an aircraft are much more stringent. Equally important is the ability to isolate or detect both permanent and transient errors soon enough to prevent data contamination which might result in erroneous aircraft commands.

The high reliability of the experimental ground-based system described here will become a requirement for nearly all ATC computers in the next two decades. Today's ATC system, which employs over 25,000 air traffic controllers, is overly labor intensive and, with the current traffic control procedures, will become even more so in the future. FAA plans call for increased automation of controller functions with a human role change from controller of every aircraft to ATC manager who handles exceptions while the computer takes care of routine ATC commands. Placing this increased responsibility on ATC computers implies that the reliability questions addressed in this paper have a much broader application than the two examples cited and are critical to the future safety of air traffic.

The most notable characteristic of the tasks enumerated at the beginning of this section is that they represent a data driven application. That is, the entire ATC process can be broken into a number of small tasks, each of which takes an entry from a list, processes the entry indepently of other tasks, and updates another list (see Fig. 1). As an example, consider the following characterization of radar surveillance processing:

o A list of digitized reply information from an aircraft is given to the computer by the radar beacon.

o Each reply is processed by an information verification task. The usual output of this task is an update to a list of tentative targets. If sufficient confidence in a tentative target exists, the task creates an entry in a list of declared targets.

o Another task takes individual declared targets and correlates them to aircraft tracks.

o A final task projects the aircraft track on the basis of past history to determine where the aircraft will be on the next rotation of the antenna.

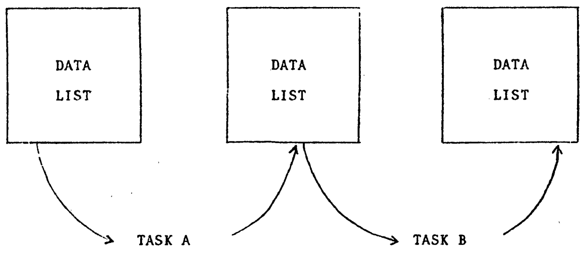


FIGURE 2: **PROBLEM CHARACTERIZATION - ATC PROCESSING IS DATA DRIVEN**

This is obviously an oversimplification and not the case for all aspects of the ATC algorithms, but experience and analysis have shown that most tasks and their data requirements do fit into this mold. Furthermore, it is generally possible to refine tasks to small modules (a few hundred instructions) when required by a specific implementation scheme.

Other important considerations are the real-time characteristics of ATC processing and the related life expectancy of the dynamic data in the system. Radar data processing is driven by two time periods, the interrogation period and the antenna rotation period. In the case of the DABS/IPC system, the antenna rotates once every four seconds and has a pulse repetition frequency (PRF) of 400 (i.e., an interrogation period of 2500 μsec). In this system aircraft are interrogated individually and thus schedules must be set up to inerrogate several aircraft during each interrogation period. While preliminary schedules are set up before the rotating antenna beam (about 2 1/2-4 degrees in width) gets near an aircraft, final scheduling cannot be performed until the aircraft actually falls within the beam. If, for example, by some preliminary processing of aircraft replies it is determined that the response from a particular aircraft was not received or was garbled (unintelligible), then the computer must schedule a reinterrogation of the aircraft before the rotating beam passes completely by the aircraft.

While response times on the order of millseconds are necessary for these beacon channel management functions, the response time for other functions in the DABS system are driven by the antenna rotation rate. Once a target is detected and a message from the corresponding aircraft is received, the system cannot communicate with the aircraft again for 4 seconds (i.e., until the aircraft falls within the beam again). During this time the aircraft track can be projected, conflicts with other aircraft determined, and an appropriate message for the next interrogation of that aircraft prepared. (An interrogation from the ground contains the message to the aircraft and the aircraft reply contains the necessary response.) Thus, the system must be able to perform the necessary tasks within 2 to 4 seconds after the arrival of the radar data.

The data in the DABS/IPC system consists of static files that represent things like the geography of the region surrounding the sensor and adjacent sensor configurations and dynamic files that contain interrogation schedules, aircraft track information, conflict list, etc. The static files are, for practical purposes, never changed and thus have an infinite lifespan. The dynamic file entries should be updated at least every four seconds on the basis of new aircraft state information. Copies that are more than 20 to 30 seconds old are no longer of interest since the relative aircraft geometry cannot be projected that far with confidence. Data in the dynamic files can therefore be said to have lifespan of under 30 seconds. These considerations are important because they have a strong influence on the memory reliability schemes for a highly reliable system.

Design Requirements

The design of a computer system for an ATC application is constrained by a number of factors relating to system reliability, software flexibility, system growth, and maintenance. This section deals specifically with the particular design requirements of an FAA operated, ground-based system like the DABS/IPC site processor but most of the constraints also apply to an airborne system. The following list summarizes the constraints:

- o Very high system reliability (20,000 MTBF)
- o Off-the-shelf CPUs, memories, etc.
- o Component standardization to facilitate maintenance
- o Low hardware acquisition cost
- o System architecture to minimize software complexity
- o Hardware and software modularity to permit:
  - addition and modification of functions at minimal cost
  - varying system size with site dependent maximum load
  - system evolution to keep pace with technology

The primary system constraints stem from the DABS/IPC 20,000 meantime between failures (MTBF) requirement. This number refers to failures from which the system cannot, through automatic switchover to an appropriate redundant element, recover within 10 seconds and without loss of data. A two hour mean-time-to-repair (MTTR) of a failed element that has been replaced by a redundant element is assumed. The 20,000 hour figure applies to a system consisting of all sensor electronics and power supplies (antenna control, receiver, transmitter, computer, and modems for intersite communication). In the case of the TI prototype design this requirement translates to a computer MTBF of over 200,000 hours.

The DABS/IPC system does not require fail-soft operation (i.e., degraded operation with a partially operational system) since the DABS/IPC network philosophy is to provide full service at all times by backing up an entire failed sensor with adjacent sensors. High software reliability is not a specific goal in the prototype system although a top-down, structured programming design approach is being used and impact on software complexity was one of the hardware system design criteria.

In the past, the approach to reliability of this order (most notably in aerospace applications) has been to use specifically designed processors (e.g., the JPL STAR computer (5), usually constructed of components with higher reliability than standard commercial components. Initial system cost constraints and FAA maintenance policies favor the use of off-the-shelf CPUs and memories configured to achieve the necessary reliability. The electronic component state-of-the-art and mini-micro processor and memory costs make this a technically viable and cost-effective alternative.

In the case of the DABS/IPC site processor the FAA expects to install from 50 to 200 systems. These vary in peak processing requirements and in site adaptation parameters. The ideal system architecture would permit deployment of computer systems of varying sizes, all capable of running the same set of software. Only one software maintenance facility and one hardware maintenance training facility should be needed. Stores for maintenance should be handled centrally. The DABS/IPC system is a part of the overall FAA ATC system and as such must be able to interface with a variety of computer systems and must be sufficiently flexible to take on new functions (or give up old functions) as other parts of the system change. The lifespan of DABS/IPC is projected to be well over 20 years and during that period the hardware will be upgraded to reflect advances in technology. This should be achieved with minimal effect on the software and without impact on continuous system operation. Finally, the FAA has expended considerable amounts of money for software development and maintenance in the past. It is envisioned that some of the results of the work of the late 60's and early 70's in the reliable, large scale software system development area will be applied to new ATC systems to safeguard against the recurrence of this state of affairs. The software should be designed to make modifications and transition to a new generation of hardware as painless as possible. This implies modularization and separation of the ATC related algorithms from architecture dependent code.

Example: Ground-Based System

Hardware

To facilitate the discussion of design considerations a description of the previously cited DABS/IPC example is in order. The DABS/IPC surveillance site processor being built by Texas Instruments consists of 20 identical DABS computers, arranged in quadrants (groups of four) and connected by TILINES [a] to two global memories of 128K words each (Figs. 2 and 3). The TILINE is a multi-user, asynchronous parallel bus capable of approximately 3 million 16-bit transfers/second. TILINES can be connected with TILINE couplers. A DABS computer is made up of two TI 990/10 CPUs, a voter, and an 8K-word 300 nsec local memory (Fig. 4). The TI 990/10 is a 3 μsec, 16-bit mini that uses register files (with 16 general purpose registers) in the local memory. Both local and global memories are single error correcting and double error detecting. The global memories are made up of 32K word modules, each with its own power supply. A global memory access holds the global TILINE for approximately 900 nsec but, because of TILINE coupler delays effective read and write times are 1200 nsec. Addressing is at the byte level. A memory map option gives an address space of 1024K words. Memory addresses are generated with a bias register (set up in software) and an offset from that bias register.

Software

Task are dedicated to computers in the DABS system. The programs and task specific data for each task are in the local memory with a copy of all programs in the global memory. Several tasks are usually put in one computer with task distribution done to balance processor load and local memory requirements. Task communication is done through global memory tables (files) with coordination achieved through semaphores. The rule that tasks may only communicate via the global memory is strictly enforced to permit task reallocation without impact on software should requirements change. If one processor is not powerful enough to handle a task this scheme makes it possible to set up two identical tasks in separate processors operating on the same input file. Although the DABS computers are ordered in priority due to physical location on TILINES, it is expected that this priority will not affect operation of the various tasks when they make global memory references. There are two reasons for this. First, the DABS computer is slow compared to global

(a) Trademark of Texas Instruments, Inc.

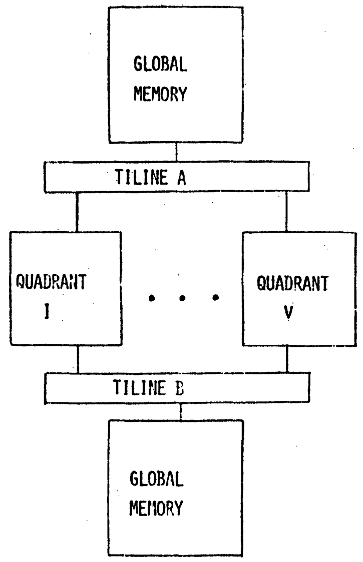


FIGURE 2 - DABS/IPC ARCHITECTURE OVERVIEW

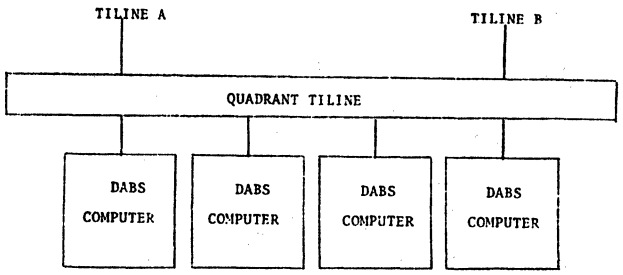
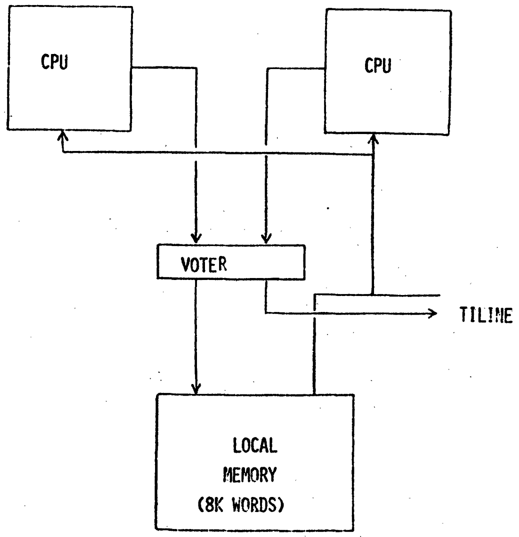


FIGURE 3 - DABS/IPC QUADRANT



CPU = TI 990/10
MEMORY = 300 nsec cycle time
single error correcting

FIGURE 4 - DABS COMPUTER

memory access time. Second, it is estimated that the ratio of local to global memory references is between 5-1 and 10-1.

The major portion of the DABS computer executive consists of simple task schedulers in each of the computers. From a philosophical viewpoint it is eminently reasonable to place the burden of deciding whether or not processing of a table entry is to be done on the individual tasks rather than on a global executive. The dedicated task scheme with semaphore coordination in a multi-mini is estimated to eliminate 75% of the executive program that would be needed for this type of application in a large-scale computer. A large part of this savings is due to the fact that in a system composed of many cheap micros (or minis) system cost is low enough so that one no longer has to worry about keeping the machine busy.

Perhaps the most difficult portion of the software design for a dedicated task, multi-mini system is the scheme for recovery from computer failures. Since temporary data in the local memory of a failed computer cannot be retrieved, the task that was interrupted must be restarted. This implies that some cleanup of global files must be performed. A semaphore scheme should be able to handle this problem, but it must be carefully designed to minimize the recovery software, particularly if there are several identical tasks. It is not anticipated that computer recovery for the DABS system will be more difficult than a checkpoint scheme for a large-scale computer.

Error Detection and Correction

CPU error detection is done on a clock-by-clock basis. The two CPUs in a computer execute identical instructions and the voter compares the output of each operation. If there is disagreement the computer is declared faulty and a spare, which may reside in a different quadrant, is switched in. When a computer failure occurs, a signal is sent to all computers and each one determines whether or not it is a spare. The first one to decide that is a spare goes into the global memory, looks in a table to decide which tasks had been assigned to the failed computer, loads the program for these tasks into its local memory, performs the previously described cleanup of pointers and begins execution.

Errors in the RAM portion of the memory are detected by the use of a modified Hamming code. If possible the error is corrected. An uncorrectable error in a local memory results in declaration of a computer failure. When an uncorrectable global memory error occurs, the appropriate 32K word memory module is declared faulty and a backup module is brought in. If the failed module contains static data, bias register tables are modified to switch all references to the backup module that contains a duplicate copy of the static data. If the failed module contains dynamic data the recovery depends on the form and method of keeping backup data.

Several alternatives for keeping backup data are being investigated at the time of this writing. One might modify the control logic so that a write command puts data into both the primary and backup module while a read command only reads from the primary module. The changeover to the backup module could then be done totally by the hardware since the address space, and thus the bias register assignment for individual files, does not change. A second alternative would be to follow every write into global memory with a second write into a backup module. Another possibility would be to take periodic snapshops of dynamic files and, if a primary module fails, continue processing with slightly aged data. Finally, one could maintain, by extra writing, sufficient up-to-date backup information to recreate the most critical dynamic files. All possibilities have serious drawbacks and thus compromise one or more of the design requirements. Special hardware is needed in one, others result in possible global TILINE contention problems or increased complexity in operational and error recovery software.

The preferred solution to global memory reliability for handling the DABS/IPC dynamic file preservation requirement is to use an error correcting memory with high enough reliability to meet the specified MTBF without the need for redundant modules. The critical element in achieving such reliability is the control logic in the memory module. In a reliability calculation for the DABS/IPC memory this contributes approximately 100 errors/10^6 hours while the RAM and power supplies (if duplexed) contribute less than 1 error/10^6 hours. The impact of this error rate is to drop the DABS/IPC system MTBF from 20,000 to 3,000 hours if dynamic memory is not backed up. It is not clear just how much the control logic reliability can be improved and thus whether or not off-the-shelf memories with the requisite reliability will become available. This and the dual-write single read memory described earlier will be investigaged. No matter which design is selected for a production system, all global memory modules will have to be the same for software flexibility and maintenance purposes.

There are several active components in the system that contribute errors which are not detected by the voting logic or the Hamming codes. The most notable are the control elements in the data paths and the memory control logic. The number of errors contributed by these elements is not large - in the DABS computer (2 CPUs, voter, local memory) for example, less than 2 percent of the errors remain undetected or, in other terms, the MTBF for the logic that may cause undetected errors is 180,000 hours. Yet, means must be provided to detect the errors, not so much for reliability purposes as to prevent data contamination. Three techniques are used: parity along data paths, periodic reads and writes into global memory, and periodic processing of known radar inputs.

Transient Errors

Since data integrity is so important in an automated ATC system, it is essential that protection be provided against transient errors. The ratio of transient to permanent errors for the particular components used is not known and, in general, data on transient rates is rare and well-guarded by manufacturers. This author suspects that for the technology used in the DABS/IPC system 80-95% of all memory and processor errors are transients. Thus, the hardware clock-by-clock checks (error detecting memory and voting CPUs) are an absolute necessity. An error detection scheme that relies heavily on software checks would be totally unacceptable. The periodic checks cited above do not protect against transients but, in the DABS/IPC system, are included for completeness to detect errors in that small portion of the logic that is not protected by other, transient detecting mechanisms. The ability to detect errors as they occur has the additional advantage of making the error correcting software much simpler.

The high ratio of transients to hard failures must be considered in designing an error recovery scheme. For detectable memory error, the normal error correcting mechanisms take care of receovery. Provisions must be made to determine whether or not a memory error was transient for maintenance purposes. If monthly maintenance is assumed, it can be predicted that all modules have had at least one transient during the month and, without knowledge of which errors are hard, all memory boards (two per module in the DABS/IPC system) would have to be replaced. For CPU errors simple computer replacement whenever a computation disagreement occurs may not be appropriate if the ratio of transients to hard errors turns out to be too high. It is predicted that a voter in a computer will detect a hard error every 5400 hours. In a system with 15 active computers, a transient to hard error ratio of 10-1 means that a backup computer would have to be brought in almost daily. To overcome this unacceptably high switch-over rate an instruction retry capability to isolate hard errors would be required. The DABS computer does not currently have such a capability.

Example: Airborne Computer

The design of an airborne computer for collision avoidance is at a much earlier stage of design than the DABS/IPC system hence a discussion of design details is not possible. The operational constraints for an airborne system are different enough from DABS/IPC to impact the overall system design and thus a brief discussion of constraints and resulting architectural considerations is included here.

Reliability is of concern in the airborne system, but the driving force is low system cost. The number of DABS/IPC in sites is in the 10's or 100's; the number of airborne systems, even if installed only by commercial air carriers, is in the 1000's. All systems will be identical and thus the size flexibility so important to DABS/IPC is of little concern here.

The proposed system again consist of a number of dedicated task processors each with a local memory and connected to a global memory. Since reliability and data integrity is not as important as low cost, CPUs will not be duplexed and voted. Cost consideration drive one toward read-only memories (ROMs) for program stores but reliability requirements may force the designer to use identical computers with local RAMs and one or more backup computers. Error correcting memory should be used for all RAMs in the system. Software considerations are quite similar to those for DABS/IPC and the same benefits of asynchronous, dedicated task design can be realized.

Discussion

Meeting Design Criteria

The DABS/IPC architecture meets most of the design criteria set forth above.

- o The specified reliability and data integrity is achieved with the clock-by-clock CPU checks and the error correcting memory.
- o While a design that requires no data duplication or memory error recovery software is preferred, the current commercial memories do not permit this. One of the goals of the current

DABS/IPC development program is to arrive at a compromise memory recovery design.

- o Executive software is simple since there is no explicit interprocessor communication and no monitor to assign tasks to processors and keep the computer busy.

- o Individual tasks can be changed, reallocated, or split between several processors with no impact on applications software. Thus, the software is flexible enough to be used in systems of differing maximum loads and to evolve as functions change or are added.

- o The overall system architecture is amenable to upgrading since the TI 990 series is an upward (or downward) compatible family of micro/minis. Use of such a family (several are now on the market) is expected to make it easier to keep pace with technology since compatible newer CPUs and memories can be plugged into the system as the old ones become outdated.

- o Maintenance is facilitated through use of commercial CPUs and global memory boards. The local memory and voter are not standard products. Maintenance software will in most cases be able to pinpoint a faulty board which can then be readily replaced and sent to a central facility for repair.

- o Initial system cost is low because standard commercial minicomputer building blocks are used almost exclusively.

Other Architectural Features

A number of algorithms in ATC radar data processing and in IPC lend themselves to special purpose processors or to microcode implementation. In surveillance processing, the most time consuming algorithms are limited in performance by memory access rates. The use of a special purpose processor to overcome this limitation is being investigated. In IPC, a heavy CPU load is imposed by the coarse screen algorithm that determines which aircraft are near one another (several DABS computers are dedicated to coarse screen). It has been shown that by implementing only a few parts of the algorithm in microcode the CPU requirement could be decreased by over 50%.

Both approaches to throughput improvement have drawbacks and must therefore be subject to careful tradeoff analyses. A special purpose processor violates the standardization requirements and will not be as amenable to a cost-conscious high-reliability implementation. Using special microcode for some of the algorithms would, in the current DABS/IPC design, require that all processors have dynamically writable microprogram memories. This adds a significant amount to the cost of a system that employs a dual CPU voting scheme. These and other variations of the current DABS/IPC architecture will be studied in terms of applicability to a production version of the DABS/IPC system and to other future highly reliable ATC computers.

Comparison with Other Multiprocessors

Many mini or micro multiprocessors have been proposed in the past and some have been successfully implemented. Two approaches that have received wide publicity are PLURIBUS (6) and the processor-switch-memory architectures of C.mmp (7) and the Burroughs D-machine (8). A brief comparison of these with the DABS/IPC system will be presented in light of the highly-reliable task oriented processing of ATC applications.

PLURIBUS is a more elegant and more complete (in a reliability sense) system than DABS/IPC. It also performs a much simpler application (all tasks are independent) and is part of an overall network that is much more forgiving than the DABS/IPC environment. PLURIBUS has a single task queue feeding all processors which normally have the same code in their local memories. (Both systems use local memories to reduce internal bandwidth requirements.) DABS/IPC tasks are much too diverse to use this type of arrangement, thus the tasks must be dedicated to identical processors. PLURIBUS can live in its network in a degraded mode of operation since that will only be reflected in a lower message throughput rate - DABS/IPC cannot perform full aircraft separation assurance unless all resources are available thus the system is designed to operate at full capacity at all times (unless of course the whole system breaks). PLURIBUS detects the effects of both hard and transient errors (and does so extremely well) with parity, checksums, timers, etc., working as part of a Concensus system. In DABS/IPC such a scheme would not work because the complexity of the overall algorithms would make error recovery too difficult if errors were allowed, as they are in PLURIBUS, to propagate into memory. On the whole, many of the architectural features of the two systems (minis with local memories, coupled bus data paths, global memories) are the same. The approaches to task allocation and to error detection and correction are different to meet different system requirements.

The two processor-switch-memory systems, unlike PLURIBUS and the DABS/IPC system, were not designed for specific applications. In particular, high reliability was not the primary design goal. Modifying these systems to achieve high reliability is more difficult than it would be to modify a bus oriented system. Although bus couplers contain active components, the expected failure rates are much lower than for a large switch. This means that a more powerful detection scheme than simple parity checks will be needed. Furthermore, the switch concept does not lend itself as readily to size flexibility as the bus-oriented processor-memory connection system. The Burroughs D-machine has no local memory (C.mmp does) and thus memory contention becomes a problem for real-time systems with fast response requirements. The D-machine does offer microprogramming (at two levels) which could achieve dramatic throughput improvements, but unless one is willing to include a large microstore, dynamic task allocation (a plus for the D-machine) would be lost through microprogram tailoring.

Conclusion

It is this author's opinion that a highly reliable system should be designed from high level elements for a specific application, as was done in the DABS/IPC system and, to a lesser extent, in PLURIBUS than to modify an existing, more general system, for high reliability. The mini-micro technology of the mid-70's is at a point where off-the-shelf processors, memories, and buses can be configured into very reliable systems at a reasonable cost and with few modifications or one-of-a-kind components.

Few high-reliability applications represent a sufficient force in today's electronic marketplace to have any effect on technology's course of development. To take advantage of low-cost elements, the designers of highly reliable systems must ride on the coattails of the whole industry and use the approach suggested above. This will lead not only to systems with low initial cost and low maintenance cost, but also to systems with an extended lifespan. As the power of microprocessors increases and as memories become faster and cheaper, these can readily replace the CPUs and memories in an existing system if the mini-micro family is properly selected.

This paper has attempted to motivate, through the use of examples, some of the desirable design features for a reliable ATC radar data processing computer to be used within the operational constraints of the FAA. It has been suggested that error detection be achieved through clock-by-clock checks (voting CPUs and error-correcting memories) and that the system architecture should be based on coupled-bus data paths. Some open questions have been raised and other approaches briefly examined for applicability to the Air Traffic Control problem.

References

(1) "A Application-Oriented Multiprocessing System," IBM Systems Journal, Vol. 6, No. 2, 1967.

(2) DABS: A System Description, Report FAA-RD-74-189, U.S. Department of Transportation, November 1974.

(3) A Description of the Intermittent Positive Control Concept, Report FAA-EM-74-1, U.S. Department of Transportation, February 1974.

(4) An Active Beacon-Based Collision Avoidance System Concept (BCAS), Report FAA-EM-75-7, U.S. Department of Transportation, October 1975.

(5) "The STAR (Self-Testing-And-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE 1971 Int. Symp. Smyposium on Fault-Computing, March 1971.

(6) Ornstein, S.M., et.al, "Pluribus - A Reliable Multiprocessor," AFIPS Conference Proceedings, Vol. 44, 1975, pp 551-559.

(7) Bell, C.G. and W.A. Wulf, "C.mmp - A Multi-Mini-Processor," AFIPS Conference Proceedings, Vol. 41. 1972, pp 765-778.

(8) Technical Summary of the Interpreter-Based System, Report TR 71-1, Burroughs Corp., 1971.

DISTRIBUTED PROCESSING FOR SIGNAL PROCESSOR USING THE BUILDING BLOCK SIGNAL PROCESSOR

Frank P. Hiner III
Litton Data Systems Division
Van Nuys, California 91409

Abstract

Requirements exist for the techniques and means wherein complex programmable signal processors may be constructed. Single computers have not near enough speed to implement signal processors, but a distributed processor approach has been determined and is being implemented.

During the last several years, the Building Block Signal Processor (BBSP) has been evolved as the processing element for a distributed processor approach to the implementation of radar and IFF signal processors. The BBSP is a high speed processor, constructed of off-the-shelf MSI and LSI and contained completely on a single circuit board.

The Remote Radar Tracking Station (RRTS) is a current example of a complex signal processing task mechanized with a set of BBSPs. The RRTS, though physically quite small, performs the automatic detection and tracking of aircraft from radar and IFF returns.

Development of the Building Block Signal Processor

The development of special purpose processors required for modern radar and IFF equipments has historically been a painful task necessitating the design and development of several smaller units of specialized digital equipment. These units are then integrated together to perform the total digital tasks. In each case, a virtually new design is required, and a new crew of engineers labors to produce the desired equipment.

The result is often a complex design which,

a. Requires arduous checkout
b. Is understood chiefly by its original designer
c. Is difficult to maintain
d. Is difficult to modify (often impossible)

Recognizing that the complete signal processor is made up of a set of functions coordinating together to perform some task (e.g., the detection of radar targets), it was a natural step to determine if a programmable processor could be evolved that would be able to implement each of the separate functions. From this position, it was one more step to seeing that some of these functions could be partitioned into subfunctions. It certainly seemed reasonable that an elementary processor could be designed that could implement the subfunctions, or major functions in many cases. A set of such processors then acting in coordination would cooperate to perform the total job.

During the design of the processing element, the following precepts and considerations guided our thinking:

a. The machine must be the processing element used to solve our signal processing design problems. Thus, during the design phase, intermediate designs were tested on paper to determine if the trial architectures were efficient at the problems we felt were most likely.

b. We desired to take advantage, to the greatest extent, of the extant MSI and LSI that were off-the-shelf and second sourced. Therefore, our architectures were influenced by the real, and soon to be real world.

c. The instruction memory and data memory were to be separate with the instructions stored in a Read Only Memory (practically speaking, this becomes PROMs).

d. The machine would use a single clock and would operate synchronously with all other like machines.

e. Recognizing that it is truly impossible to cover all jobs with a single design, consideration was given to the requirement to interface special purpose hardware to the processor to solve certain complex tasks. For example, if a very high speed FFT is required, then a high speed butterfly might be designed that would be controlled by a BBSP.

f. The machine was not to be in any way a general purpose computer and hence complex I/O structures and multi-level interrupts would not be required.

The resulting design effort produced the Building Block Signal Processor (BBSP) in the spring of 1972. The machine was first incorporated into a delivered equipment in the late fall of that year. In the next several years, several versions of the BBSP were produced. During this time, the programming language we had produced (BUBAL) also evolved.

Two significant hardware changes occurred in these years. First, we added a return address stack, allowing nested subroutines and interrupts. Second, printed circuit board technology had developed to the point where multilayer printed circuits for very dense boards were possible allowing us to package a complete 12 bit BBSP on a single 8" x 8" circuit board. A system could now be designed as a distributed processor utilizing the BBSP as the processing element with automatic maintenance to the single card level easily attainable. For the designer, the single card machine meant that if another function had to be added to the machine, or the implementation of a present function became too difficult, he could simply add another BBSP to the system with small impact on system size (take two, they're small).

Description of the BBSP

The 12 bit BBSP (see Figures 1 and 2) is completely contained on a single 8" x 8" standard logic board and is constructed of off-the-shelf low power Schottky devices. With these devices, it has a power consumption of 12 to 15 watts, and operates at a speed of 2 MIPS (i.e., clock interval of 500 nano-seconds). The board contains a switch, lamp, and special op code for self-test. Upon pressing the switch, the error lamp is lit and the machine goes into its self-test using a portion of the wired in memory. The self-test must turn the lamp off for the machine to be demonstrably error free.

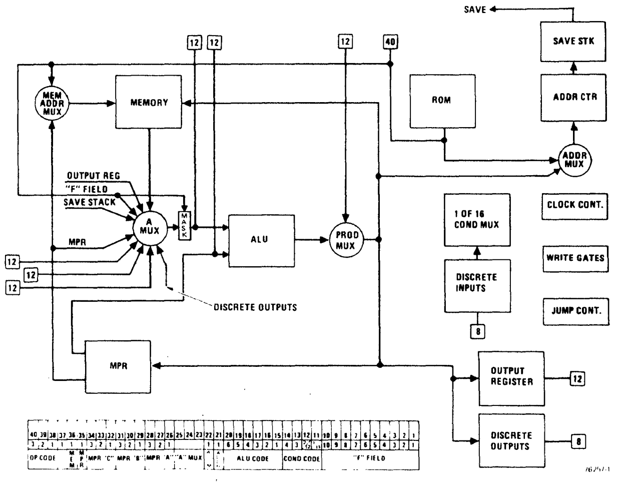


Figure 1. Block Diagram of the BBSP

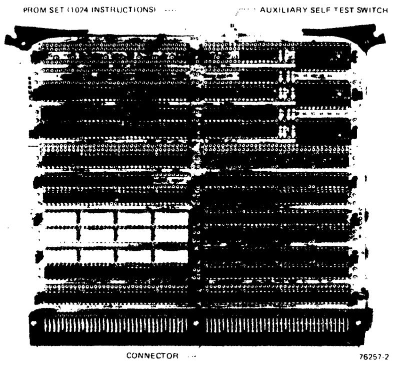


Figure 2. The BBSP Card

A 1024 word instruction Read Only Memory is addressed by the instruction counter. This counter advances sequentially in binary fashion unless:

a. An interrupt occurs which forces the address to zero.
b. A JUMP occurs, which loads the literal field (F-Field) from the instruction into the address counter.
c. A RETURN occurs, which loads the output of the PRODUCT MUX into the address counter.

The output of the Instruction Memory passes directly to the machine. All instructions (except double precision operations) including a satisfied JUMP, RETURN or CALL take one clock period. Interrupts and CALLS are allowed by virtue of a 16 level RETURN address stack.

Data is stored in the 1024 word data memory and the 8 word Multi-port register (MPR). The data memory may be addressed by the literal field or by any of the words in the multi-port register. Outputs from the BBSP occur via the 12 bit output register and the 8 discrete output flip-flops.

A left MUX allows the left argument of the ALU to be the data memory, one of three external inputs, an MPR, the output register, the F-Field or the return address stack. This MUX may be masked by the F-Field. The ALU arguments are brought to the connector to be used as the arguments for an externally located function (such as a multiply). The output of this function is brought back into the card, through the connector, into a product mux whose other input is in the ALU output. The product mux feeds the memory, MPR and output register.

The machine allows two kinds of double precision. There exists double precision instructions that allow a 24 bit sum or difference to be calculated in 2 consecutive clock intervals. Additionally, one may parallel two machines (cards) to obtain 24 bit operation in a single clock.

Program Checkout

To enable programs to be checked out without burning an endless number of PROMs, Loadable Instruction Memories (LIM) cards have been designed that are compatible with the BBSP. A BBSP and a LIM card are plugged in side by side and wired together on the back plane. The LIM additionally contains electronics to facilitate the testing of programs. Each LIM communicates data serially to a single Programmer's Control Unit. This unit allows the programmer to inspect the memory contents, output register, discrete outputs, or product MUX output of any one of the BBSPs in a system up to a maximum of 16. From this unit, the programmer may stop the system clock, advance it by single step or stop it on an address match. He may also inspect memory contents and change memory contents.

Program loading is performed via a paper tape reader. Programs are kept on a disc at a time sharing computer along with the Assembler. Assemblies result in a listing and a paper tape.

Once a program is finalized for a BBSP, the last paper tape is used directly to program the PROM. The PROM chips are mounted on a single removable circuit board which itself plugs into the BBSP card. When the PROM module is plugged in, the LIM card is removed from the system leaving a plenum behind.

The Remote Radar Tracking Station – An Example of a Distributed Processor Using the BBSP

In the following paragraphs, details are given on the implementation of the Remote Radar Tracking Station (RRTS), a distributed processor utilizing the BBSP as the processing element (see Figure 3). The RRTS is the latest in a sequence of evolutionary steps which began with the large SAGE radar data processors of the 1950s.

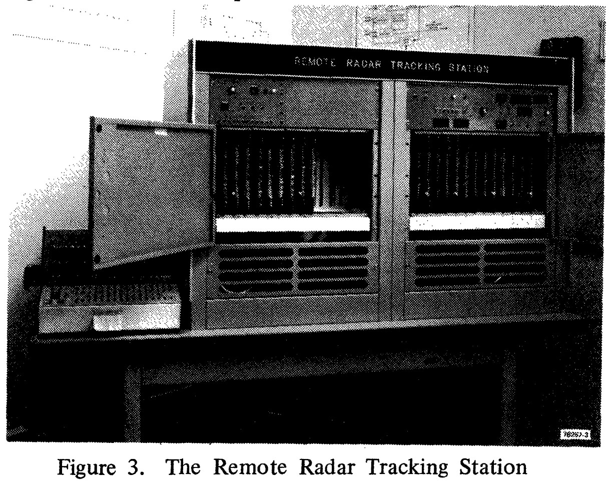


Figure 3. The Remote Radar Tracking Station

The RRTS has as its task totally unattended automatic radar and IFF target detection and tracking. This processing must be performed automatically and unattended in the presence of normal radar receiver noise and radar clutter (i.e., returns from objects other than aircraft, such as weather, land masses, automobile, etc.).

Target position, velocity and IFF code data obtained from the RRTS are transmitted through a communication link over standard 1200/2400 BPS modems to remoted users. While all pulse search radars can benefit from such technology, the gap filler and remote radar applications are immediate beneficiaries.

The basic RRTS interfaces with a wide variety of radar pulse search radar types accepting synchro data, resolver data or ACP/north mark azimuth data. The RRTS accepts the radar's normal and MTI videos, forms a fine grain clutter map (currently 65,000 cells) and automatically switches between normal and MTI video. The clutter map and an MTI residue mapper also work together to determine when small localized regions of the surveillance region must be censored due to bad MTI performance (such regions otherwise produce excessive false alarms). The outputs of the radar detection logic are applied to the tracker unit which processes up to 256 targets, plus a false alarm burden, in an automatic track initiate and track update mode. A feedback path exists from the tracker to the clutter mapper and residue mapper preventing valid targets from being mistaken for clutter.

The tracker automatically controls IFF interrogations in SIF/MARK XII. The responses are detected, decoded, and correlated with the radar returns before being input to the tracker. The outputs of the system communicate through a communications processor to the remote user.

A local maintenance monitor continually tests the system and triggers an audible alarm and visual indicators locally upon a fault being detected. The detected faults also cause the comm link to be shut down if the trouble is pathological. The local maintenance man initiates the Automatic Troubleshooting sequence which then locates the failed card with a MTTR of one minute to a confidence level of 95 percent.

Mechanization of the RRTS

Figure 4 is a block diagram illustrating the partitioning of the tasks of the RRTS into relatively disjoint subtasks. Some of these jobs are performed in pipeline fashion and some in parallel with other jobs in the machine. Some of the jobs operate in real time and others in near real time. It should be reemphasized that radar processors have been built in the past many times, containing many of the functional units here shown but that in these applications all of the units, excluding the tracker and the comm processor, have been implemented with special purpose (hardwired) logic.

RRTS Partitioning Notes

The partitioning of this machine into processing elements follows fairly naturally from the radar/IFF processing tasks which must be performed. Special purpose machines of the recent past have employed a similar partitioning. Once the major functional areas have been identified, the system designer must determine an algorithmic approach using the building block elements to implement the function. On our initial look at this machine, two BBSPs were designated as required each, by the Automatic Clutter Mapper (ACM), the tracker, the digital radar detector and the IFF processor. All other areas were felt to require a single BBSP. As system design proceeded several algorithmic tricks reduced the ACM BBSP count to one. Analysis of worst case target and fruit rates combined with algorithm advances reduced the IFF processor BBSP count to one. Similarly, the tracker BBSP count reduced to one. The Communications Processor increased to two.

Several units had additional functions added to them rather than add machines. The Azimuth Converter (AC) for example, receives all system switches through a set of multiplexers (which it addresses) and was given the task of performing the system initialization. The CFAR unit which had been loafing, was given the additional task of computing the system Maximum Range (RMAX) trigger based on the synchronized trigger obtained from the Quan/Rang sync card. A portion of the radar detection logic, which had a low duty cycle, was given the additional task of correlating the radar and IFF reports.

Detailed Example: The Radar Detection Logic

The radar detection logic is here defined as that logic which accepts the output of the video-to-digital quantizers, a range counter, and azimuth counter and produces a low false alarm rate message locating the target in range and azimuth. This message is then transferred to the tracker.

The digital detector uses two levels of quantization (see Figures 5 and 6) and makes use of the shape of the returns in azimuth, which is a function of the antenna beamshape. The algorithm first quantizes the video into three levels and then time quantizes the outputs into range bins.

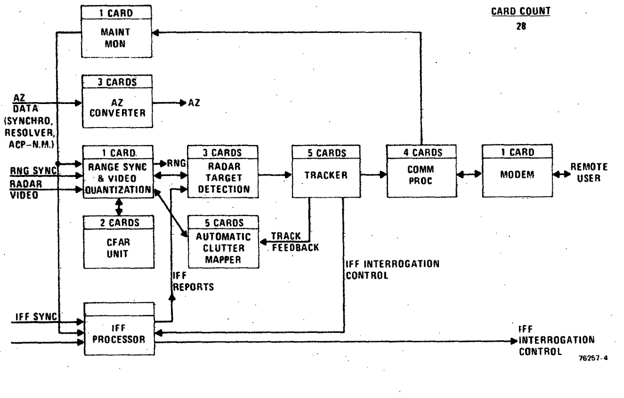
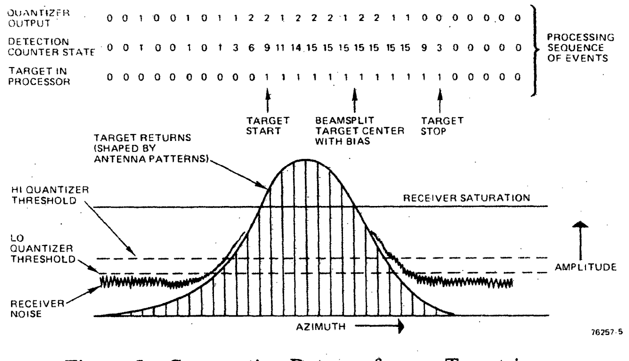


Figure 4. Functional Block Diagram of the RRTS



QUANTIZER OUTPUT 0 0 1 0 0 1 0 1 1 2 2 1 2 2 2 1 2 2 1 1 0 0 0 0 0 0
DETECTION COUNTER STATE 0 0 1 0 0 1 0 1 3 6 9 11 14 15 15 15 15 15 15 15 9 3 0 0 0 0 0
TARGET IN PROCESSOR 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
PROCESSING SEQUENCE OF EVENTS
TARGET START
BEAMSPLIT TARGET CENTER WITH BIAS
TARGET STOP
TARGET RETURNS (SHAPED BY ANTENNA PATTERNS)
HI QUANTIZER THRESHOLD
RECEIVER SATURATION
LO QUANTIZER THRESHOLD
AMPLITUDE
RECEIVER NOISE
AZIMUTH
76257-5

Figure 5. Consecutive Returns from a Target in a Single Range Cell

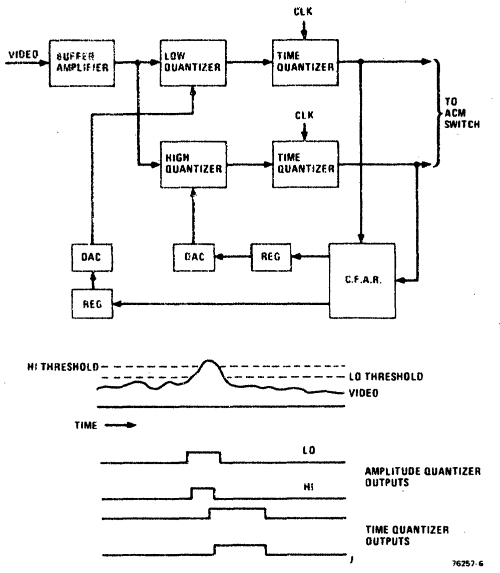


Figure 6. Digitization of the Radar Video

For each range bin, a four bit counter is maintained (see Figure 7). This count is set as a function of the previous state of the counter and the present output of the quantizers. The function is implemented by a state matrix which is a look up table with the counter and quantizer output being the address of the table and the contents of the table being the next counter value. The matrix is designed to respond to the edges of the beam pattern. The matrix is a function of the number of hits per beamwidth. A total of 6 state matrices have been designed using simulations to empirically jointly optimize detection ability and azimuth accuracy. The final phase of the detection logic is to measure the distance between the start and stop signals and to guarantee that this distance is greater than a minimum established by the beamwidth; if so, a target is declared and the center azimuth is computed.

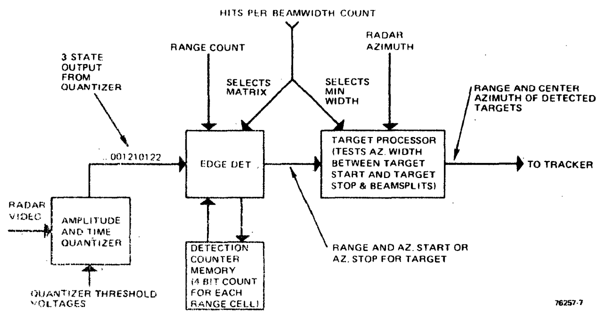

HITS PER BEAMWIDTH COUNT
RADAR AZIMUTH
RANGE COUNT
3 STATE OUTPUT FROM QUANTIZER
SELECTS MATRIX
SELECTS MIN WIDTH
RANGE AND CENTER AZIMUTH OF DETECTED TARGETS
001210122
EDGE DET
TARGET PROCESSOR (TESTS AZ WIDTH BETWEEN TARGET START AND TARGET STOP & BEAMSPLITS)
TO TRACKER
RADAR VIDEO
AMPLITUDE AND TIME QUANTIZER
DETECTION COUNTER MEMORY (4 BIT COUNT FOR EACH RANGE CELL)
RANGE AND AZ START OR AZ STOP FOR TARGET
QUANTIZER THRESHOLD VOLTAGES
76257-7

Figure 7. Basic Digital Detector Block Diagram

Implementation of this function with BBSPs (see Figure 8) takes cognizance of the false alarm rates of the processes. A four bit counter must be maintained for each range cell and tests continually made on the cell contents. A single BBSP (the edge Detector) is employed for this task and continually loops through a highly efficient program making one pass through the program loop for each range cell.

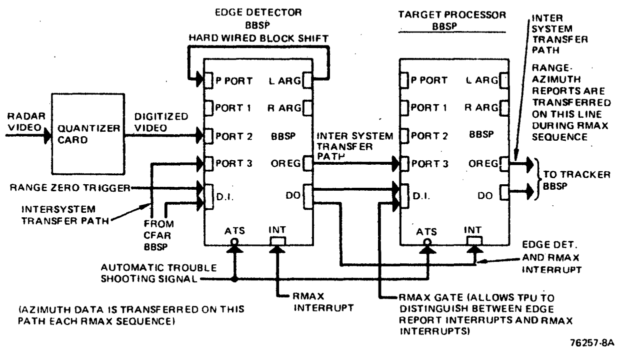

EDGE DETECTOR BBSP
HARD WIRED BLOCK SHIFT
TARGET PROCESSOR BBSP
INTER SYSTEM TRANSFER PATH
RANGE-AZIMUTH REPORTS ARE TRANSFERRED ON THIS LINE DURING RMAX SEQUENCE
RADAR VIDEO
QUANTIZER CARD
DIGITIZED VIDEO
P PORT
PORT 1
PORT 2
PORT 3
D.I.
L ARG
R ARG
BBSP
OREG
DO
ATS
INT
INTER SYSTEM TRANSFER PATH
TO TRACKER BBSP
RANGE ZERO TRIGGER
INTERSYSTEM TRANSFER PATH
FROM CFAR BBSP
AUTOMATIC TROUBLE SHOOTING SIGNAL
EDGE DET. AND RMAX INTERRUPT
(AZIMUTH DATA IS TRANSFERRED ON THIS PATH EACH RMAX SEQUENCE)
RMAX INTERRUPT
RMAX GATE (ALLOWS TPU TO DISTINGUISH BETWEEN EDGE REPORT INTERRUPTS AND RMAX INTERRUPTS)
76257-8A

Figure 8. Implementation Diagram of the Digital Detection Function

The start and stop signals from the edge detector are fed to a second BBSP, named the Target Processor Unit (TPU), which holds the start report, with its range and azimuth until a stop report is received and then tests the difference azimuth and calculates the center azimuth. As start and stop signals from the edge detector occur at a relatively low rate, the TPU can operate in non-real time. Thus, the start and stop signals from the Edge Detector interrupt the TPU and load the report with its range into an input queue. Azimuth is obtained during the RMAX sequence (see below) once, each sweep. The TPU then in non-real time brings the reports out of the input queue, compares them with reports stored in memory and performs the minimum width measurement and azimuth computation. Once a target is declared, it is put into an output queue (in BBSP memory) and transferred, during a subsequent RMAX sequence, to the tracker unit.

Intersystem Communication

Design of the communication and synchronization of the individual processing elements in the RRTS is a function of the timing characteristics of pulse search radar sets and the rate of data exchange between the defined subfunctions of the RRTS (see Figure 9). Radars are synchronized by a timing trigger which defines the transmitter firing rate (typical range: 200 to 1000 pulses per second). A range maximum trigger can be defined that occurs at the end of the radar listening time.

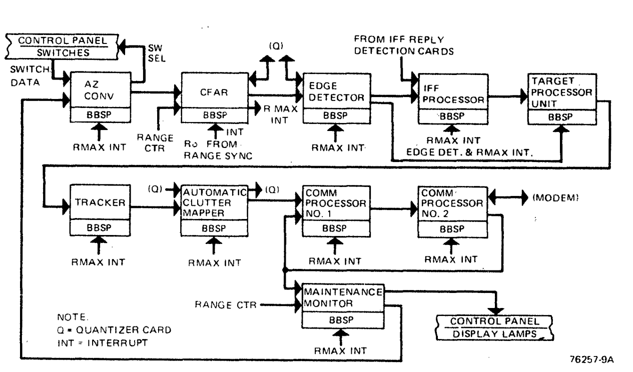


Figure 9. RRTS Inter BBSP Communication

The RRTS accepts the radar range zero trigger and during system initialization calculates a range maximum trigger (RMAX) to be used thereafter. The RMAX trigger is then generated internally by the RRTS each listening time and interrupts all BBSPs in the machine. Following this interrupt, an intersystem transfer occurs (the RMAX sequence). A transfer path has been designed into the system such that each BBSP acts as a node in a directed graph connected as a continuous loop. Common system data is passed throughout the system in bucket brigade fashion.

These transfer paths are also used in initialization and switch monitoring. Virtually all control panel switches are input through a multiplexer to a single port on the AC BBSP. During system initialization, this unit calculates common system constants and distributes these constants throughout the system via the loop transfer path. Additionally, the AC BBSP reads a different control panel switch each RMAX time and then distributes the switch code and the switch value throughout the system. As all BBSPs in the system receive this information, each BBSP checks for specific switch value changes of interest to itself and modifies its operation accordingly.

Intersystem Synchronization

Units such as the TPU in the radar detection logic, the IFF processor, the tracker and the Comm. Processor operate in non-real time on inputs received from the real time units. Once initialized, they remain effectively quiescent until receiving input. The real time units are synchronized by the range zero signal. Units such as the ACM and the Edge Detector utilize endless program loops that are exited only upon receipt of the RMAX interrupt. These units then perform the RMAX transfers, change any parameters as necessary and then wait for the range zero trigger. The range zero trigger, synchronized to line up the system clocks by the Quan/range sync card, then times these units into their real time loops.

Fault Detection

System fault detection is performed by the maintenance monitor. Periodically, the maintenance monitor injects targets into the front end of the system and monitors system outputs for correct operations. During the RMAX sequence, advantage is taken of the loop nature of the transfer path to check that basic intersystem transfers and program execution are occurring as normal. Other checks are made on the CFAR, Az converter, ACM and tracker to monitor numerical quantities which are indicative of system normal operation.

Maintenance

Upon a fault being detected, audible and visual alarms are enabled on the control panel. The local maintenance man may then, by switch action, initiate the Automatic Trouble-Shooting sequence (ATS). This causes the maintenance monitor to first perform a self test, and if this is successful, to transmit the ATS strobe to all BBSPs in the machine. Each BBSP contains a common ATS program which performs an exhaustive diagnostic program on itself. Upon the ATS strobe being received, the FAULT lamp on each of the BBSPs is illuminated and the machine executes the self-test program. The FAIL lamp is extinguished if and only if the machine passes the test. Upon a BBSP passing its test, it has the responsibility for checking out its dedicated peripheral cards (e.g., in the case of the Tracker, it has three mass memory cards and a multiple array card to be checked out). During the ATS mode several of the discrete outputs of the BBSPs are defined as error lamp control for a specific peripheral card. To test a specific unit, the BBSP executes a diagnostic program, and if it finds an error, turns on the discrete output. This discrete output, in ATS mode only, causes the FAIL lamp to glow on the tested peripheral. If no error is detected, the BBSP proceeds on to test the next peripheral until all have been tested or an error has been found.

As a backup to this automatically initiated and controlled system, each BBSP is provided with an ATS switch mounted on the card itself. At any time, the curious can press this switch and initiate the internal ATS program (and the ATS program of the associated peripherals, if any) and then observe the FAIL lamp(s). This does disrupt the system, obviously, and the system must then be reinitialized.

Summation and Conclusions

A distributed processor approach to the design of radar and IFF signal processors has been determined using the BBSP. The RRTS has been designed and implemented using this approach. The RRTS is consequently a programmable and highly modifiable signal processor, a combination that until now has not occurred. The processing element used is the BBSP which is contained completely on a single 8" x 8" board. Four card types, a mass memory card, a multiply array and two special purpose cards, were used along with the BBSP to implement each of the subfunctions of the RRTS. As the BBSP and these ancillary cards are logic functions, each completely contained on a single board, the system could be and was designed with the ability from day one to be able to locate failures to the single card level to a confidence level of 95 percent with an MTTR of one minute.

Several techniques were developed which generalized the machine and greatly sped development.

a. All processing elements are a node in a single continuous data loop.

b. Transfers from one BBSP to the next in the data loop are effected by a single control signal allowing data to be passed along in bucket brigade fashion.

c. All instructions take exactly one clock time and all machines run off of the same clock.

d. All control panel switches are multiplexed into a single BBSP which then distributes the switch values to the rest of the machine on the data loop.

e. Each BBSP contains a diagnostic program which completely tests itself and its dedicated peripheral cards.

The distributed processor approach to the RRTS implementation greatly shortened design time, reduced the number of required design engineers and produced a system which can be modified extensively by programming changes.

Acknowledgements: The conceptualization of the distributed processing approach to radar processing was derived jointly with Mr. Mike Tolle. BBSP development owes much to the labors of Mr. George Simms while the current RRTS hardware is largely due to the efforts of Messrs. Fred Erickson, Sam Pawley and Al Kushida. The digital detection algorithms described owe their paternity to Dr. Don Adelman.

ASSOCIATIVE-PARALLEL APPLICATIONS TO RADAR SIGNAL PROCESSING

K. L. Schaffer
Hughes Aircraft Company
Fullerton, California 92634

Abstract -- Two aspects of radar signal processing that are excellent candidates for associative/parallel implementation are spatial correlation and adaptive processing. By performing the correlation with a high degree of parallelism to yield short processing times, the data can be processed more than once using a variable detection logic to maximize resource utilization without saturation of the system. Additionally the process can be preceded by adaptive filter weight control to minimize the desensitization due to high amplitude clutter or interference. Parallel processing can be used to significantly speed up the convergence of this form of adaptability over normal hardware implementations. Joint implementation of these processes by using associative/parallel mechanizations can significantly increase radar effectiveness in heavy interference environments.

CORRELATION PROCESS

Highly sensitive doppler radars must perform detection over frequency and range domains separated into small resolution cells. When ambiguities arise in one of these domains due to, for example, high Pulse Repetition Frequency (PRF) operation, the detection processing becomes heavily cluttered with false correlations. Associative/parallel processing can alleviate this problem and provide an inherent form of adaptability by nature of its processing speed.

Classically, radar detection is based on a Neyman-Pearson detection criteria using the several variables typically available in the form of azimuth, elevation, range and frequency coordinates. Several correlations within these domains are usually utilized to exploit the differences in expected target behavior versus the behavior of noise or unwanted interference. This is accomplished by filtering and determining the mean level signals in the vicinity of the cell of interest with the resulting filtered signal to mean (S/M) ratio being the fundamental statistic to be dealt with. This represents the starting point of the following discussion of a detection process using associative/parallel processing for beam to beam and range cell to range cell correlation for each doppler filter.

Detection Word Formulation

In order to minimize the number of bits to be included in memory for this type of detection, a multiple threshold technique is used. In the case here, two thresholds are used as shown in Figure 1 with the threshold values matched to the expected target occurrence. I.e

$$\frac{T_1}{T_2} = \left[\frac{p(\Delta\phi)}{p\ (o)}\right]^2$$

where

$\Delta\phi$ = beam spacing

$p(\Delta\phi)$ = azimuth (or elevation) pattern with a peak at p(o)

In this manner, the probability of a low threshold adjacent beam crossing given that a target has crossed a high threshold center beam is nearly unity whereas for noise only the joint probability of two crossings is low. Thus, by creating a 2 bit detection word within the target word, the stage is set for beam to beam correlation. The assignment rule is as follows:

$$d_{ij} = \begin{cases} 1 & \text{if } V_{ij}/\overline{M}_{ij} > T_1 \\ 2 & \text{if } V_{ij}/\overline{M}_{ij} \geq T_2 \\ 0 & \text{otherwise} \end{cases}$$

where

d_{ij} = detection bit

i denotes the ith elevation beam

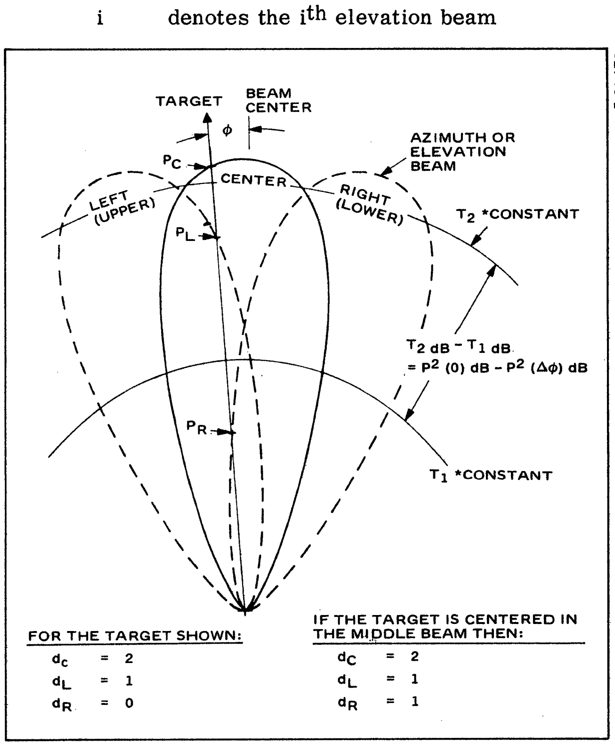


Figure 1. Multiple Threshold Detection

j	denotes the jth azimuth beam
V_{ij} =	Stored voltage magnitudes
$\overline{M}_{ij}$ =	Mean Level
T_1 =	Low threshold
T_2 =	High threshold

Additional logic is incorporated in the beam to beam correlations to determine when the target return exceeds T_2 on several beams which would result in multiple reports of a single target.

<u>Beam to Beam Correlations</u>

The detection word formed in the previous step can now be utilized to perform beam to beam correlations. The basic correlation matrix is a 9 beam configuration (3 azimuth x 3 elevation beams). Therefore, define a 3 x 3 matrix as follows:

$$B = \text{Beam Correlation Matrix} = \left[b_{ij}\right]$$
$$= \text{Expected outcome for a centered target}$$

and B is of the form

$$b_{ij} = b_{ji} \text{ and } b_{i1} = b_{i3}$$

Example:

$$B = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Now let D be the matrix of detection words that are actually measured.

$$\begin{array}{cc} \xrightarrow{\text{Azimuth}} & \\ D = \begin{bmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{bmatrix} & \Bigg\downarrow \text{Elevation} \end{array}$$

Define Y, by Y = DB and let

$$y = \sum_i \sum_j y_{ij} = \text{decision Statistic}$$

Then

$$y = \sum_i \sum_j \left(\sum_{k=1}^{3} d_{ik}\, b_{kj} \right)$$

which, for the stated properties of B, yields

$$y = (2\,b_{11} + b_{12}) \sum_i (d_{i1} + d_{i3}) +$$
$$(2\,b_{12} + b_{22}) \sum_i d_{i2}$$

or

$$y = C_1 \sum_i (d_{i1} + d_{i3}) + C_2 \sum_i d_{i2}$$

where

$$C_1 = 2b_{11} + b_{12} \text{ and } C_2 = 2\,b_{12} + b_{22}$$

For the example given previously, $C_1 = 1$ and $C_2 = 4$

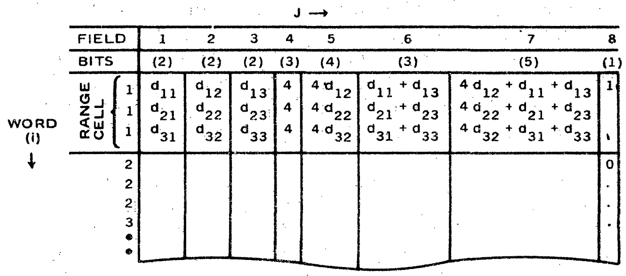
The decision rule is

if $y \geq To$; a target is present

if $y < To$; no target is present

(To = 13 for the example problem)

Figure 2 shows the implementation for a single filter and range cell and Figure 3 shows the potential power of a fully parallel implementation to handle several filters simultaneously. The process is basically a sliding window in azimuth with each elevation scan of three beams moving to the left as a new set is loaded in. Thus the result of each decision refers to the beam occupying the center location and if a detection is made the detection word is set to one; otherwise it is set to zero.


J →
WORD (i) ↓
RANGE CELL

FIELD	1	2	3	4	5	6	7	8
BITS	(2)	(2)	(2)	(3)	(4)	(3)	(5)	(1)
1	d_{11}	d_{12}	d_{13}	4	$4\,d_{12}$	$d_{11} + d_{13}$	$4\,d_{12} + d_{11} + d_{13}$	1
1	d_{21}	d_{22}	d_{23}	4	$4\,d_{22}$	$d_{21} + d_{23}$	$4\,d_{22} + d_{21} + d_{23}$	
1	d_{31}	d_{32}	d_{33}	4	$4\,d_{32}$	$d_{31} + d_{33}$	$4\,d_{32} + d_{31} + d_{33}$	'
2								0
2								.
2								.
3								.
•								
•								

(1) $(F2_i) \rightarrow (F1_i)$; FOR ALL i

(2) $(F3_i) \rightarrow (F2_i)$ FOR ALL i

(3) NEW ELEVATION SCAN INTO $(F3_i)$ FOR ALL i

(4) $(F1_i) + (F3_i) \rightarrow (F6_i)$ FOR ALL i

(5) $(F4_i) * (F2_i) \rightarrow (F5_i)$ FOR ALL i

(6) $(F5_i) + (F6_i) \rightarrow (F7_i)$ FOR ALL i

(7) $\sum_{i=k}^{k+2} (F7i) \rightarrow (F8_k)$ k = 1, 4, 7, 10 ...

(8) IF $(F8_k) > T_o$, THEN SET THE DETECTION BIT TO 1: OTHERWISE SET 0

(NOTE: FJ_i denotes the ith word of the Jth field)

Figure 2. Detection Process for Beam Correlations

In order to illustrate this process, assume two targets are present resulting in a data set as shown in Figure 4 (two adjacent targets in azimuth beams 2 and 3 and one in number 6 with all targets somewhere between elevation beams 1 and 2). As can be seen, even in heavy traffic, the basic resolution of the radar is maintained. Note that in this simplified example no advantage is taken of the off axis quantities that may be present if the beams are closely spaced (this is a result of $b_{11} = b_{13} = b_{31}\ b_{33} = 0$). Hence the example is not optimized and is approximately equivalent to

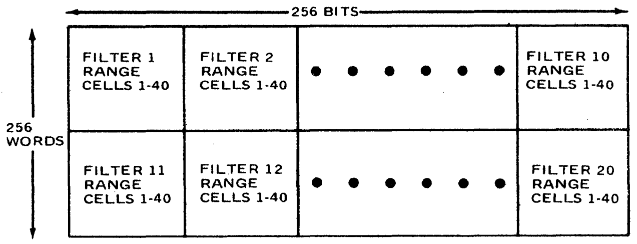
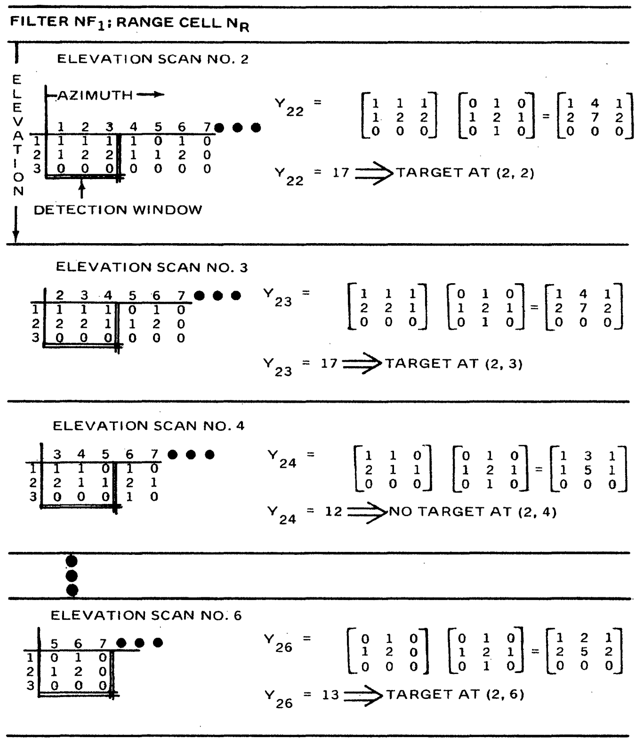

256 BITS
256 WORDS
FILTER 1 RANGE CELLS 1-40
FILTER 2 RANGE CELLS 1-40
FILTER 10 RANGE CELLS 1-40
FILTER 11 RANGE CELLS 1-40
FILTER 12 RANGE CELLS 1-40
FILTER 20 RANGE CELLS 1-40

Figure 3. Multiple Filters Can Be Processed Simultaneously



$$Y_{22} = \begin{bmatrix}1&1&1\\1&2&2\\0&0&0\end{bmatrix}\begin{bmatrix}0&1&0\\1&2&1\\0&1&0\end{bmatrix} = \begin{bmatrix}1&4&1\\2&7&2\\0&0&0\end{bmatrix}$$

$$Y_{22} = 17 \Longrightarrow \text{TARGET AT (2, 2)}$$

$$Y_{23} = \begin{bmatrix}1&1&1\\2&2&1\\0&0&0\end{bmatrix}\begin{bmatrix}0&1&0\\1&2&1\\0&1&0\end{bmatrix} = \begin{bmatrix}1&4&1\\2&7&2\\0&0&0\end{bmatrix}$$

$$Y_{23} = 17 \Longrightarrow \text{TARGET AT (2, 3)}$$

$$Y_{24} = \begin{bmatrix}1&1&0\\2&1&1\\0&0&0\end{bmatrix}\begin{bmatrix}0&1&0\\1&2&1\\0&1&0\end{bmatrix} = \begin{bmatrix}1&3&1\\1&5&1\\0&0&0\end{bmatrix}$$

$$Y_{24} = 12 \Longrightarrow \text{NO TARGET AT (2, 4)}$$

$$Y_{26} = \begin{bmatrix}0&1&0\\1&2&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&1&0\\1&2&1\\0&1&0\end{bmatrix} = \begin{bmatrix}1&2&1\\2&5&2\\0&0&0\end{bmatrix}$$

$$Y_{26} = 13 \Longrightarrow \text{TARGET AT (2, 6)}$$

Figure 4. Example of the Beam Correlation Process

treating the elevation and azimuth correlations separately. In the event the beams are closely spaced it is an easy matter to include these terms in the constants in order to achieve improved detection performance. Generally the constants can be represented by factors of two and hence the multiplications can be carried out in an expedient manner.

Range Correlation

The detection word resulting from the beam correlations can now be used to perform range correlations and resolve the true target range. The most straight forward way to accomplish this is to expand the measured ambiguous range interval by replicating the detection word into locations based on the following rule:

$$i = i' + k_j N_j;\ 1 \le i' \le N_j;\ k_j = 0, 1, 2 --- (K_j - 1)$$

where

- i = target range cell (true location)
- i' = measured location
- N_j = Number of unambiguous range cells for the jth PRF
- K_j = Total number of unambiguous range intervals for the jth PRF contained from 0 to the maximum range.

Once this is accomplished for 3 PRF's, the detection bits can be added for all range cells in parallel with a detection declared in a range cell only if the sum exceeds a specified value as indicated in Figure 5 for a simplified case where $N_1 = 7$, $N_2 = 6$ and $N_3 = 5$.

FILTER 1:

	1	2	3	4	5	
	Ri	PRF1	PRF2	PRF3	Σ	
	1	1	0	0	1	
	2	0	1	0	1	
	3	0	0	1	1	
	4	0	0	0	0	
N_3	5	0	0	0	0	
N_2	6	0	0	0	0	
N_1	7	0	0	0	0	
	8	1	1	1	3	TARGET
	9	0	0	0	0	
	10	0	0	0	0	
	11	0	0	0	0	
	12	0	0	0	0	
	13	0	0	1	1	
	14	0	1	0	1	
	15	1	0	0	1	
	16	0	0	0	0	
	17	0	0	0	0	
	18	0	0	1	1	
	19	0	0	0	0	
	20	0	1	0	1	
	21	0	0	0	0	
	22	1	0	0	1	
	23	0	0	1	1	
	24	0	0	0	0	
	25	0	0	0	0	

(1) LOAD DETECTION BITS FOR ALL THREE BEAMS (3 PRF's)

(2) $(F2i) \rightarrow (F2_{i+kN_1})$ ALL i, k=1, 2 ... $(k_1 - 1)$

(3) $(F3i) \rightarrow (F3i+KN_2)$ ALL I, k=1, 2 ... $(k_2 - 1)$

(4) $(F4i) \rightarrow (F4i+KN_3)$ ALL i, k=1, 2 ... $(k_3 - 1)$

(5) $\sum_{J=2}^{4} (FJi) \rightarrow (F5i)$

(6) IF $(F5i) \geqslant 3$, THEN SET DETECTION BIT TO 1; OTHERWISE SET 0.

Figure 5. Range Correlation Using Multiple PRF's

With this arrangement, only targets with detection on all three PRF's will be retained, which due to independence of the noise samples, will reduce the false alarms while simultaneously determining the true target range. Some false targets, however, may be retained due to either clutter residue or correlations of noise with images of the target. The severity of this depends on the number of false targets contained at the input of the range correlation process.

This process is configured in a similar manner to the beam correlation process in that basic processing section consists of all the range cells for a given filter and the cells are processed simultaneously. Several filters can also be processed simultaneously for a high degree of parallelism depending on the size of the Associative Processor (AP) dedicated to this task.

Optional Adaptive Feature

The target load passed on to the tracking computer is largely dependent on the environment when fixed detection logic is employed. This is true even with mean level detection when the statistical nature of clutter is other than Rayleigh. This is a likely occurrence in the presence of ground clutter. Therefore, in a heavy clutter environment, the system may be saturated with false alarms while in a clear environment the system may only be used to 10 or 20 percent of its capacity. In fact both of these conditions can occur within a single scan in a long range air surveillance radar.

The detection process utilized here can correct this situation if a high degree of parallelism is used to yield high processing rates. This is done in an iterative fashion by performing the beam and range correlations and then counting the number of detections. This number is compared with the number of new target reports the tracking computer is willing to accept. If the number is excessive, the process is repeated with a more stringent detection logic (e.g. higher thresholds). If the number of targets is too low, which implies the system sensitivity is not being exploited, the process is repeated with lower thresholds. Thus with two or three iterations the system sensitivity can be matched to the requirements of the tracking computer and the available resources will be utilized to maximum capability on a beam by beam basis.

ADAPTIVE FILTERING

The intent of this investigation is to develop an adaptive algorithm to be used by a parallel processor with emphasis on taking advantage of the programmable parallel structure. Typically the required mathematics utilizes matrix manipulations and often the resulting formulations, though theoretically solvable, are not practical when time, cost, and size are considered. Thus the approach here is to first define the theoretical solution and then reduce it to an approximation that is feasible for implementation. The basic approach to be utilized is an iterative technique with the gain function in the iterative relation being the processor's primary means of dealing with radar clutter.

Defining The Problem

The following defines the notation to be used and briefly states the problem at hand:

Let

$X(k)$ = received signal incident on the filter elements from the kth range cell (a vector quantity)

Thus

$$X(k) = \begin{bmatrix} x_1(k) \\ x_2(k) \\ \cdot \\ \cdot \\ x_n(k) \end{bmatrix} = \begin{bmatrix} c_1(k) + n_1(k) \\ c_2(k) + n_2(k) \\ \cdot \\ \cdot \\ c_n(k) + n_n(k) \end{bmatrix}$$

Where

$c_i(k)$ = clutter voltage

$n_i(k)$ = noise voltage

Let

$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \text{element weights}$$

and

Let $y(k) = \Sigma w_i^* \, x_i(k) = W^+ X$ (+ denotes complex transpose)

$y(k)$, then, is the scalar output of the filter

Also let $P = XX^+$

and $\overline{P} = \text{AVG}\,[XX^+]$ = covariance matrix

Assuming $c(t)$ is wide-sense stationary, $\overline{n_i \, n_j^*} = \delta_{ij}$

and $c_i \, n_j^* = o$ for all i and j, we get

$$\overline{P} = \begin{bmatrix} Rc(o)+Rn(o) & Rc(T) & Rc(2T) \ldots Rc[(N-1)T] \\ Rc(-T) & Rc(o)+Rn(o) & Rc(T) \ldots \\ Rc(-2T) & Rc(-T) & Rc(o)+Rn(o) \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ Rc[-(N-1)T] & & Rc(o)+Rn(o) \end{bmatrix}$$

Where

$T = 1/PRF$

$Rc(T) = \text{AVG}\left[c(t)\, c^*(t+T)\right]$

$Rn(o) = No = \text{AVG}\left[n^2(t)\right]$

In terms of these definitions, then, one can show for a large class of optimization functions the desired answer for the weights is given by

$$\boxed{W = \overline{P}^{-1} S}$$

Where

$$S = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix} = \text{Steering vector}$$

(This is the location of the peak response, e.g., if $S_i = (-1)^i$, the peak response of the filter is at PRF/2 in the frequency domain)

The problem, therefore, becomes one of finding $\overline{P}^{-1}$ and hence the amount of effort required for a solution depends on the properties of this matrix function.

The primary characteristic that creates a problem occurs when heavy clutter is encountered. For this case, $\overline{P}$ is nearly singular and $\overline{P}^{-1}$ becomes difficult to find. This will become apparent in later quantitative treatments of $\overline{P}$.

<u>An Iterative Approach To Finding $W=\overline{P}^{-1} S$</u>

In order to gain some insight into the nature of the iterative procedure and the influence of $\overline{P}$, it is initially assumed that $\overline{P}$ is known and that W is to be found. This can be accomplished as follows:

Let

$$W(k+1) = W(k) + G\left[S - \overset{*}{Y}(k)\, X(k)\right]$$

And

$$W(o) = o \quad (G = \text{NXN Gain matrix})$$

In general, k is the kth iteration but since our concern is to process within a short time interval it is assumed the clutter statistics are wide-sense stationary over the entire unambiguous range and hence k is the kth range cell. (If clutter mapping is available, k could be the kth scan). Letting $Y(k) = W^{+}(k)\, X(k)$, noting that $Y^{*}(k) = X^{+}(k)\, W(k)$ and further assuming that the W's vary slowly relative to the X's and taking the expected value yields

$$W(k+1) = W(k) + G\, S - \overline{P}\, W(k)$$

To simplify the analysis, this can be transformed to a normal coordinate system where scalars instead of vectors can be studied. Therefore, let M (modal matrix) be defined by

$$M^{-1} \overline{P}\, M = E$$

where

$$E = \text{eigenvalues of } \overline{P} = \begin{bmatrix} e_1 & o & o & \cdots \\ o & e_2 & & \\ o & & \ddots & \\ \vdots & & & e_n \end{bmatrix}$$

Furthermore, let

$$W = MW'$$

$$S = MS'$$

$$G' = M^{-1}\, GM$$

Then

$$W'(k+1) = W'(k) + G'\left[S' - E\, W(k)\right]$$

If G is constrained to be diagonal, this yields

$$w_i\,(k+1) = w_i\,(k) - g_i\left[s_i - e_i\, w_i\,(k)\right]$$

where the lower case letters indicate scalars. One can now use z - transforms to establish that

$$w_i\,(k+1) = \frac{s_i}{g_i} - \frac{s_i}{g_i}\,(1 - g_i\, e_i)^k$$

$$= \text{Steady state} + \text{transient}$$

If

$$\left|[1 - g_i\, e_i]\right|_{\max} < 1, \text{ then}$$

$$w_i = \lim_{k\to\infty} w_i(k+1) = \frac{s_i}{g_i}$$

or equivalently

$$W' = \lim_{k\to\infty} W(k) = E^{-1}\, S'$$

$$M^{-1} W = E^{-1}\, M^{-1} S$$

$$W = ME^{-1}\, M^{-1} S = \overline{P}^{-1} S$$

Thus with the above constraint, the desired result is achieved. The question now is how many samples are required.

Let

$$\rho\,(i) = \left|\text{Transient/steady state}\right| = \left|1 - g_i\, e_i\right|^k$$

For narrowband clutter this becomes

$$e_1 = 1 + N\gamma;\ \gamma = \text{clutter to noise ratio (CNR)}$$

$$e_i = 1 \text{ for } 2 \le i \le N$$

If $g_i = g$ for all i (a typical hardware case), then

$$\rho(1) = |1 - g(1+N\gamma)|^k$$

$$\rho(i) = |1 - g|^k \quad \text{for } 2 \le i \le N$$

Now define ρ_c = measure of transient performance and

Let

$$\rho_c = \left\{ \text{MAX} [\rho(i)] \right\}$$

Ignoring the effects of loop noise for the time being, this yields a choice for g such that

$$-1 + g(1+N\gamma) = 1 - g$$

$$g = \left(\frac{2}{2+N\gamma}\right)$$

(i.e., we have minimized ρ_c)

and thus

$$\rho_c = \left(1 - \frac{2}{2+N\gamma}\right)^k = \frac{N\gamma}{2+N\gamma}$$

which for large γ becomes

$$\rho_c \cong \left(1 - \frac{2}{N\gamma}\right)^k$$

Setting $\rho_c = 0.37$ yields

$$\frac{1}{\ln\left(1 - \frac{2}{N\gamma}\right)} = \text{ko} = \text{required number of independent samples}$$

or

$$\boxed{\text{ko} \cong \frac{N\gamma}{2}} \quad \text{(Coupled Loops)}$$

Thus for large γ, the convergence is quite slow.

The case just presented represents an implementation where the correlation processes for each element uses identical fixed gains. If one could choose the g_i''s independently, then the choices would be

$$g_1' = \frac{\epsilon}{1+n\gamma}\ ; \ (1-\epsilon) << 1; \quad \epsilon \text{ is a parameter to be chosen}$$

$$g_i' = \epsilon \quad \text{for } 2 \le i \le n$$

and the transient term would be $(1-\epsilon)^k$ which is independent of γ, and for suitable choice of ϵ, yields a fast convergence. ϵ cannot be chosen equal to unity due to loop noise effects which will be discussed next.

LOOP NOISE CONSIDERATIONS

The steady state output power for the iterative procedure given in the previous topic is given by

$$Po' = \frac{S^+P^{-1}S}{S^+S} \left[1 - 1/2 \Sigma g_i e_i\right]$$

The best result that is achievable theoretically is given by

$$Po = \frac{S^+P^{-1}S}{S^+S}$$

Thus a degradation given by the bracketed term is experienced due to "hunting action" of the iteration. In more familiar terms, there is a loss in output S/(C+N) ratio of (1+K) where $K = 1/2 \Sigma g_i e_i$. For the two previous cases this would yield a degradation of

$$2 \text{ for } g = \frac{2}{2+N\gamma}$$

and

$$1 + \frac{N\epsilon}{2} \text{ for independent choices}$$

Thus in the latter case one cannot choose $\epsilon = 1$ since excessive loop noise would result. With this in mind and recognizing the ultimate goal is to achieve detection of a target, let K be specified and identical in both cases.

Under this constraint it can be shown the g_i's are given by

$$g_c'(i) = \frac{2K}{N \sum_1^N e_i} \quad \text{for the constant } g' \text{ case (coupled loops)}$$

and

$$g_u'(i) = \frac{2K}{N e_i} \quad \text{for independent choices of } g'_i\text{s (uncoupled loops)}$$

where K = specified degradation factor. This yields transient responses expressed in independent samples given by

$$\boxed{\begin{array}{ll} k_{oc} = \frac{N\gamma}{2K} & \text{(coupled loops)} \\ k_{ou} = \frac{N}{2K} & \text{(uncoupled loops)} \end{array}}$$

For

$$K = 1/4 \text{ (1 dB of degradation)}$$

$$k_{oc} = 2N\gamma$$

$$k_{ou} = 2N$$

Thus in heavy clutter, the uncoupled case greatly accelerates the convergence to the desired solution.

Preliminary Development Of The Algorithm

Up to this point, the solution has been carried out in a normal coordinate system which simplifies the problem once the transformation is accomplished. Thus a choice of coupled or uncoupled system is more complex to implement than one may initially realize. The real problem is in achieving the uncoupled system since the coupled approach is already the simplest (though least effective) approach.

Since the G' matrix is diagonal with at least two different values along the diagonal for the uncoupled case, the G matrix (original coordinates) is in general a NXN matrix. In hardware terms, this means that output of every element is available for multiplication by constant and summation at the output of every other element. Alternatively, one could perform the actual transformation and work in the transformed domain. The transforming network is also quite complex.

An associative parallel processor however, can be configured in a manner such that either operation can be achieved by allocating the appropriate AM (Associative Memory) size to this task. Since this allocation and also the implementation can be software controlled, a great deal of flexibility can be achieved.

Even with the advantages of a parallel processor the actual transformation is a time consuming process. The order of events is as follows.

(1) Acquire a dwell

(N-PRI'S x m Range cells)

(2) Estimate $\overline{P}$

$$\frac{1}{mN}\Sigma\Sigma\,(I^2 + Q^2) \text{ and } \frac{1}{mN}\Sigma\Sigma\,\mathrm{TAN}^{-1}\frac{Q}{I}$$

(3) Find the eigenvalues of $\overline{P}$

(solve an N^{th} order polynominal)

(4) For each eigenvalue find

Adj $[\lambda_i I - \overline{P}]$ and select a non-zero column. (Adj = adjoint operator)

(5) Form M (Modal Matrix) from step 4 and carry out the transformations

Even with routines that combine the above operations, a significant amount of time can be consumed in steps (3) and (4) for large N and $\overline{P}$ nearly singular. Thus an alternate approach is desirable.

Recalling that even with uniform gains, the steady state answer is achieved in the limit as $k \to \infty$; the problem is not whether adaptivity is taking place but rather how fast is it happening? Therefore, rather than jumping to the ultimate approach, a compromise approach can be utilized. This involves solving for M apriori in closed formed as a function of the quantities in step 2. This of course is restricted to cases where a closed form M can be found and the ultimate success depends on how close the form of the assumed modal matrix agrees with the actual one. In any event, it appears likely the results will be better than a totally coupled system.

Assume, therefore, that a closed form expression for M can be found and call it $\hat{M}$. Define the transformations in a similar fashion as before but using $\hat{M}$ instead of M.

$$\hat{M}^{-1}\,\hat{P}^{-1}\,\hat{M} = \hat{E} \text{ (This relation defines } \hat{P})$$

$$W(k) = \hat{M}\,W'(k)$$

$$S(k) = \hat{M}\,S'(k)$$

and

$$G(k) = \hat{M}\,G'\,\hat{M}^{-1}$$

The desired answer in transformed coordinates is

$$W'(k-1) = W'(k) + \frac{2K}{N}\,\hat{E}^{-1}[S' - \hat{M}^{-1}\,P\,\hat{M}\,W'(k)]$$

or

$$W(k+1) = W(k) + \frac{2K}{N}\hat{M}\,\hat{E}^{-1}[\hat{M}^{-1}S - \hat{M}^{-1}\,P\,W(k)]$$

$$= W(k) + \frac{2K}{N}\,\hat{M}\,\hat{E}\,\hat{M}^{-1}\,[S - P\,W(k)]$$

Note however that one can define a $\hat{P}^{-1} = \hat{M}\,E\,\hat{M}^{-1}$

Thus

$$\boxed{W(k+1) = [I - \frac{2K}{N}\,\hat{P}^{-1}\,P]\,W(k) + \frac{2K}{N}\,\hat{P}^{-1}S}$$

(recall $P = XX^+$ and is the quantity measured by the radar.)

Thus any clutter model for which $\hat{P}^{-1}$ (rather than $\hat{M}$) can be expressed as a closed form expression of γ and ϕ, can be used to generate an approximately uncoupled, iterative relation without steps (3) and (4).

The case used here is for narrowband clutter (i.e., $R(T') \cong \gamma$ over the interval $(N+1)\,T$ where $T = 1/PRF$). With some manipulation one can show that

$$\hat{P}^{-1} = [P_{ik}]\,(\frac{1}{1+N\gamma})\,; \quad ([P_{ik}] \text{ denotes matrix P})$$

where

$$P_{ik} = \begin{cases} 1 + (N-1)\,\gamma \text{ for } i = k \\ \mathrm{Exp}\,[-j\,(i-k)\,T/\tau \bullet \phi] \text{ for } i \neq k \end{cases}$$

γ and ϕ are found as in step (2) and are available as a result of radar measurements (ϕ is actually the intersample phase shift) and T/τ = number of unambiguous range cells

The above procedure was utilized in a simulation and indeed converged very rapidly when the data used to generate ϕ was known exactly. However, to be realistic, ϕ was estimated after complex noise samples were introduced and the resulting errors in ϕ caused the results to fluctuate excessively. The iteration procedure was re-evaluated with the following result:

The approximation of the phase term contains an error component due to noise. This inturn causes the eigenvalues of iteration to be of the form

$$e_i \cong 1 - \frac{2k}{N}(1 + N\gamma\sigma_\Phi)$$

where

$$\sigma_\Phi = \frac{\alpha}{\sqrt{mN}}$$

m = number of range cells in the sample space

α = function of the estimation procedure ($\alpha = 2$ for first forward difference estimates to of ϕ)

This yields a degradation factor of

$$K' = \frac{2K}{N}\left[1 + \alpha\sqrt{\frac{N\gamma}{m}}\right]$$

and thus to return to the desired specified degradation ($K' = K$), the gains must be reduced by

$$1 + \alpha\sqrt{\frac{N\gamma}{m}} .$$

This was inserted in the simulation and had the desired effect of smoothing the filter output.

The above relation was established on an approximate analysis bounding the desired result and may not yet be the optimum answer but it suffices for the time being. For example, a reduction in gain of

$$(1 + \alpha\sqrt{\frac{\gamma}{m}})$$

also smoothed the filter output until wideband clutter was inserted. In this case it appears that there is a relation between the gain adjustment and the clutter bandwidth give by

$$\Delta G = (1 + \alpha\sqrt{\frac{N'\gamma}{m}})^{-1} \cdot \frac{2K}{N}$$

Where

N' = spectral spread in filter bandwidths

In any event, a choice of $N' = N$ does not slow down the convergence appreciably and is the safest choice without apriori information on the clutter. Thus the relationship actually implemented in the simulation is

$$\boxed{\begin{array}{l} W(k+1) = [I - \Delta g\,\hat{P}^{-1}P]\,W(k) + \Delta g\,\hat{P}^{-1}S \\ \text{where} \\ \Delta g = \frac{2K}{N}\left(1 + \alpha\sqrt{\frac{N\gamma}{m}}\right)^{-1} \end{array}}$$

Results

A computer simulation of narrow and broadband clutter was developed to test the validity of the rapid convergence algorithm. The results are shown in Figures 6 to 8. Figures 6 and 7 are included to indicate nominally what happens when the filter is required to simultaneously reduce the clutter response while maintaining a peak in the specified direction. In fact, the nominal response indicated in Figure 6 is the direct result of the choice of the steering vector (S). Figure 7 indicates the adaptation of the filter for narrowband clutter, which in this case results in a null at the clutter location in doppler. For wideband clutter, the null is shifted somewhat towards the highest sidelobe so that the smaller sidelobe can cancel the impact of the higher one. (The two adjacent sidelobes are out of phase).

Figure 8 shows the improvement factor for several different cases as a function of the number of samples. Note that the rapid convergence algorithm (uncoupled loops) does converge much faster than the coupled cases. The limitation on the broadband case is a result of the choice of S and could be improved by tapering "S" to yield lower nominal sidelobes and hence lower residual clutter power. In all cases though the coupled loops yield rapid convergence.

Summary And Conclusion

The results are very promising; particularly in terms of the capabilities of the postulated associative processor. Recall however, that samples are taken over the range domain (i.e., identical filter weights at all ranges) and hence some nice assumptions about the range correlation of the clutter have been implicitly included. One could conceivably postulate clutter statistics unsuitable for these algorithms. Therefore more effort is required for specific applications. However, since the uncoupled approximation converges very rapidly, it is less dependent on the assumptions and offers a higher probability of success in real environments. For example, if 10 range cells are needed for convergence, the range domain can be divided into 10 cell increments and identical weights would apply over 10 cells rather than the entire range domain.

The primary problem of slow convergence rate, then, has been greatly reduced and in this regard the analysis using AP technology has been successful. In addition, a partial solution to the broader problem of realizable adaptive filter benefits in actual environments has also been achieved by virtue of the more rapid convergence.

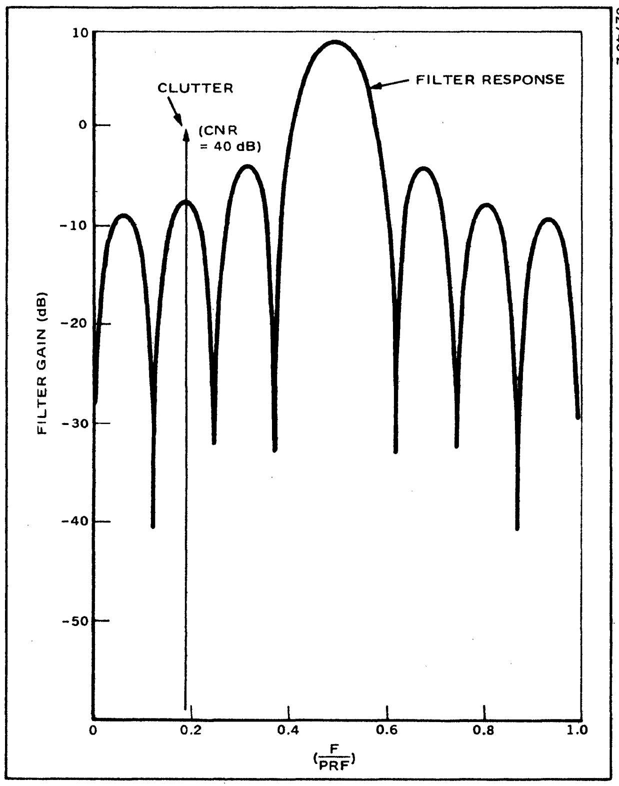


Figure 6. Nominal Filter Response

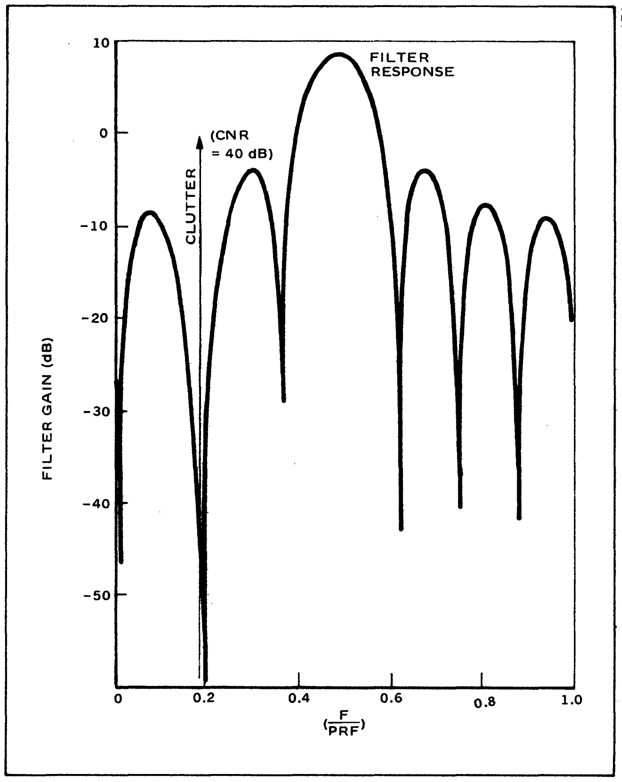


Figure 7. Adapted Response

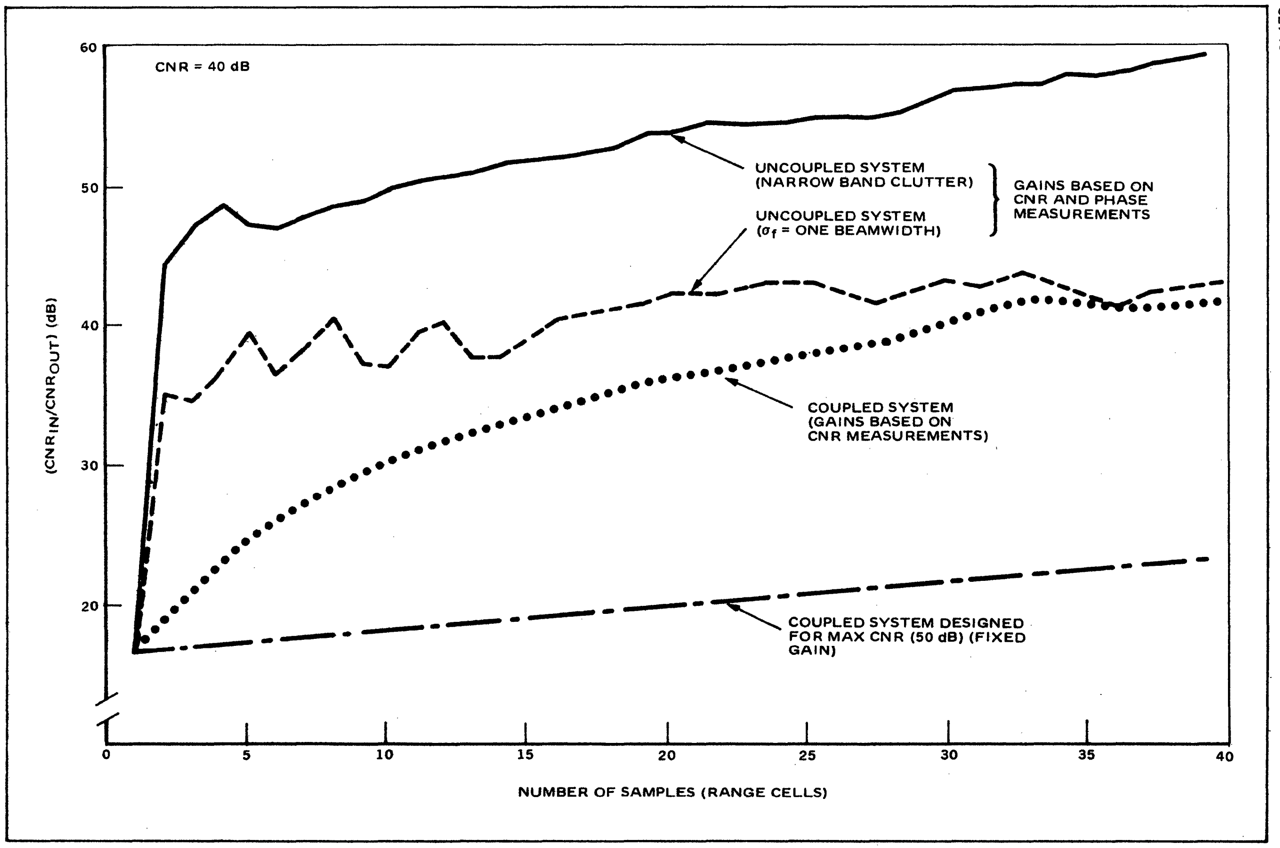


Figure 8. Comparison of Approaches

A RECEIVER FOR PCM CODED DIGITONE AND MF SIGNALS USING ASSOCIATIVE PROCESSING

Eugene S.Y. Shew and Jack M. Cotton[a]
Bell-Northern Research
Ottawa, Canada

Abstract -- The techniques of discrete Fourier transform (DFT) for the detection of digitized signalling frequencies in telephone signalling systems shows a great deal of inherent parallelism which is well suited to implementation using associative processing techniques. This paper reviews the background of DFT and the characteristics of tone signalling in telephony and develops algorithms for the parallel calculation of signalling frequency power spectra. The calculations are suitable for implementation on a purposely designed associative processor. Some estimates are presented to show that a signalling receiver based on this technique is practicable.

1. INTRODUCTION

The Digitone[b] and MF (multifrequency) receiver is an electronic filtering device tuned to a number of discrete frequencies in the voice band with the ability to determine the presence of the two strongest frequencies within a prescribed period of time and from which to determine a pre-arranged code. It is used in telephone switching systems for the decoding of Digitone digits and other supervisory signals.

In the conventional telephone offices where switching is analog, the signal receiver usually consists of a number of bandpass filters coupled to some electronic or electro-mechanical logic [1]. This paper, however, deals with receiving such signals in a pulse code modulation (PCM) time-division-multiplex exchange by means of digital filtering.

A number of receiver techniques for digital signals have been advanced, such as the digital counter technique [2]. These suffer from the common defect of not being programmable. A programmable digital MF signal receiver using discrete Fourier transform has been demonstrated [3] to be practicable to implement using special-purpose sequential hardware. The basic discrete Fourier transform approach is adopted here not only because it is well proven, but also because the algorithm shows a great deal of inherent parallelism amenable to associative processing.

Designing the receiver out of custom designed associative processing cells [4] seems to give a number of advantages. First, a modular design would provide flexibility in configuring various exchange office sizes. Second, it would be adaptive to other frequency sets on either a permanent or programmable basis. Third, the expected increase in processing speed would make it possible to handle higher data rates.

[a] Now with ITT/TTC, Stamford, Connecticut.
[b] Trademark of Northern Telecom Limited for Dual-Tone Multifrequency Signalling.

2. INPUT DATA

The Digitone frequencies consist of eight well-defined audible frequencies normally originating from a Digitone telephone set. As shown in Figure 1, four frequencies represent the columns and the other four represent the rows. Hence, activating a key transmits two frequencies to the switching centre. Although the fourth column is normally not present in a telephone set, it is nevertheless used in other signalling devices (e.g., voice response systems) and the 1633 Hz frequency is recognized. In addition, the receiver needs to detect the two dial tone frequencies for power comparison. Thus, Digitone frequency analysis involves 10 frequencies.

Multifrequency pulsing (or MF) is a method for inter-office communication. Signals such as trunk switching, calling number forwarding, and call supervision are transmitted and received

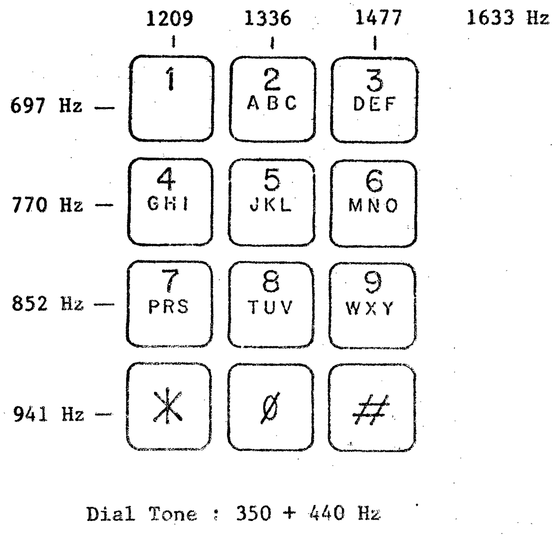


Fig.1. The Digitone and Dial Tone Frequencies

using two out of six frequencies in the voice band: these are 700, 900, 1100, 1300, 1500, and 1700 Hz. These signals may come through the same paths as the Digitone frequencies, but the two sets of frequencies do not mix within an analysis period.

3. INPUT SIGNAL CHARACTERISTICS

An analog signal of voice or combined signal frequencies is first sampled to produce PAM (pulse amplitude modulation) pulses. Twenty-four channels are multiplexed to produce interleaved PAM pulses at discrete time intervals. The encoder converts the magnitude of each PAM pulse into an 8-bit code. This conversion causes round-off errors which result in what is known as quantizing noise.

The effect of quantizing noise can be reduced by increasing the number of quantizing levels and by a coding technique known as companding [5]. The 8-bit companded code format is shown in Figure 2 where the sample is represented by a sign bit, a 3-bit exponent of base 2 (L), and a 4-bit mantissa (V). Companding provides smaller quantizing steps in the range where signal probability is high, and larger quantizing steps where signal probability is low.

As the first design objective and for compatibility, the receiver will interface with T1 trunks which have a format of 24 channels per frame, and 8 bits per channel with a data rate of 1.544 MHz or 125 microsec sampling rate per channel.

4. SYSTEM CONFIGURATION

Three criteria dictated the receiver design: the analysis algorithm, which will be discussed later, the sampling rate, and the partial result accuracy necessary to achieve a high detection probability. The proposed design consists of a basic 12 by 12 array sufficient to handle two channels in the 125 microsec sampling period and to provide 16-bit accuracy, making it a dual receiver. As shown in Figure 3, n receivers are controlled by a common control and a shared program store. The ROM (read only memory) feeds the receiver with pre-defined constants peculiar to the set of frequencies being analyzed. The output is either a digit or a null.

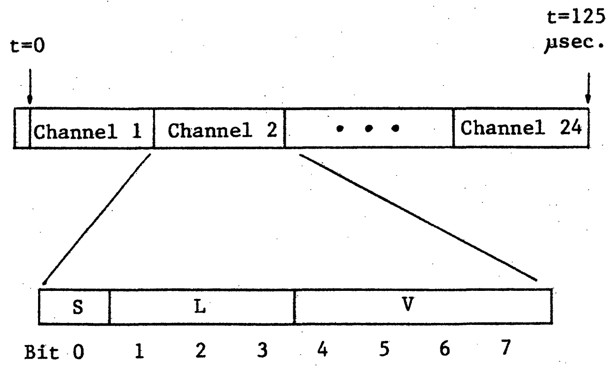


Fig.2. Data Format on T1 Trunk

5. STEPS TO SOLUTION

The following analysis centres around the Digitone detection because it is more complex than the MF, but the algorithm is the same.

The simplified flowchart in Figure 4 shows the necessary iterative functions which must be performed in real time and whose implementation will be discussed in detail later. The final computation and the rest of the digit recognition logic is not critical in terms of time or storage and requires no novel schemes, we do not intend to deal with its implementation in this paper.

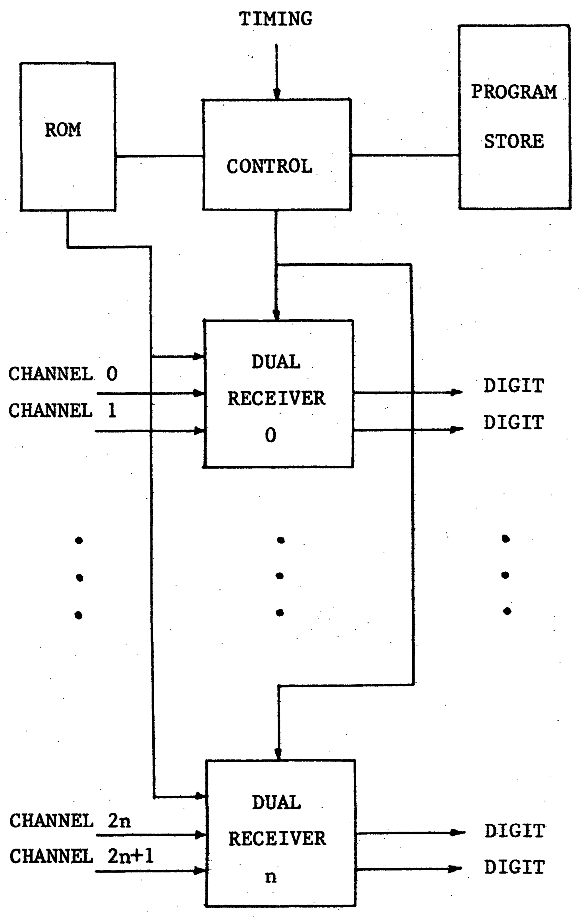


Fig.3. System Organization

A. Decompanding and Windowing

The first step is to decompand (or expand) the 8-bit sample value to its 12-bit linear form by applying the following simplified equation:

$$\text{Linear Word} = 2^{L}\,(V+16.5)$$

where:	L	is the 3-bit exponent
	V	is the 4-bit mantissa
	16.5	pertains to the characteristic of the quantization.

Successful analysis of the samples requires that they be properly windowed in order to present a well-defined frequency spectrum to the analysis program. The window [6] consists of a series of weighting constants, one for each sampling period. There is one window of 160 samples for Digitone equivalent to 20 msec, and another window of 80 samples for MF equivalent to 10 msec. Thus each sample would be weighted by a unique window constant before it is analyzed. The telephony standard specifies that a valid Digitone signal must be greater than 40 msec;

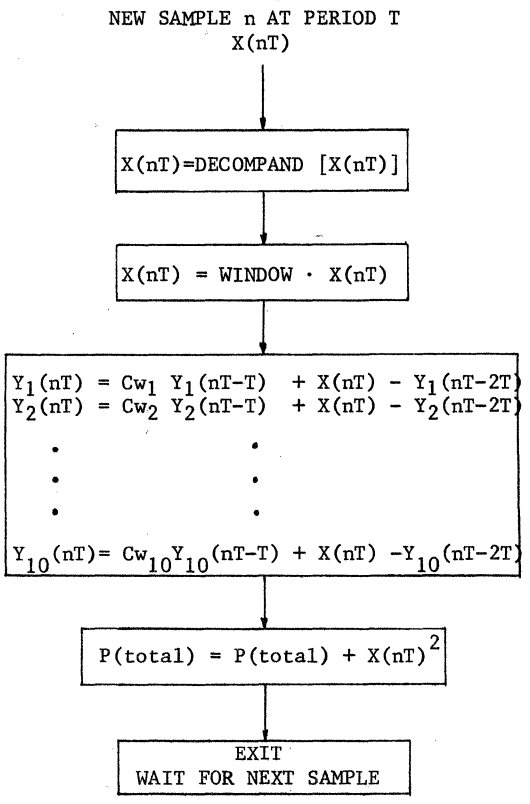

NEW SAMPLE n AT PERIOD T
X(nT)
X(nT)=DECOMPAND [X(nT)]
X(nT) = WINDOW · X(nT)
$Y_1(nT) = Cw_1\, Y_1(nT-T) + X(nT) - Y_1(nT-2T)$
$Y_2(nT) = Cw_2\, Y_2(nT-T) + X(nT) - Y_2(nT-2T)$
$Y_{10}(nT) = Cw_{10} Y_{10}(nT-T) + X(nT) - Y_{10}(nT-2T)$
$P(\text{total}) = P(\text{total}) + X(nT)^2$
EXIT
WAIT FOR NEXT SAMPLE

Fig.4. Goertzel Iterative Computation for $n = 0,1,2, \ldots ,N$ Samples

hence, a 20 msec analysis interval ensures at least one try on the shortest possible Digitone signal. The time specification for MF is derived in a similar manner.

B. Short Term Power Spectrum

The basic approach is to compute the short term power spectrum over the frequency band desired, then locate the two frequencies having the highest powers. If they are from the two orthogonal groups, the digit can be determined.

The power spectrum computation is done by discrete Fourier transform and is the critical path of the frequency analysis. The two well-known ways of performing a discrete Fourier transform, namely the fast Fourier transform (FFT) and the Goertzel algorithm, both exhibit highly parallel mathematical manipulations. Comparitive studies have shown that for the detection of k frequencies over N samples, FFT requires N words to do $2\log_2 N$ multiplications and $3\log_2 N$ additions; while Goertzel needs k words to do (N+1) multiplications and 2(N+1) additions. Thus although FFT is faster, Goertzel requires much fewer words in our receiver application. Furthermore, an extra step of frequency interpolation will be required in the case of Digitone if the result is derived from FFT. Therefore Goertzel gives a better speed/storage trade off.

The Goertzel algorithm has two parts: the iterative computation and the final computation. The iterative equation is:

$$y(nT) = Cw.y(nT-T)+x(nT)-y(nT-2T)$$

where:	$x(nT)$	is the sample at time nT
	$y(nT)$	is the interim output at nT
	$y(nT-T)$	is the output of the previous sample
	$y(nT-2T)$	is the output of the next previous sample
	Cw =	$2\cos(wt)$ for frequency w.

For n samples, this equation iterates n times and there would be one equation for each frequency coefficient, Cw, 10 for Digitone and 6 for MF. Since each iteration involves a new sample and depends on the result of the last two samples, the associative processor must be able to complete all equations within one sample period.

C. Final Computation

The final computation of the Goertzel algorithm is done only once. It is:

$$Y(nT) = -\exp(-jwt).y(nT-T)+y(nT)$$

where:	$-\exp(-jwt)$	is the complex coefficient for frequency w
	$Y(nT)$	is the final complex power output.

This computation and others can be performed at the end of the windowing period.

6. ASSOCIATIVE IMPLEMENTATION

Implementation of the iterative equation is not straightforward in a small associative array such as we propose, if we are to make best use of the space available. Functions have to be built up by a hierarchy of macros starting from the primitive micro-instructions.

A. A Simplistic Approach

A simple associative processing (AP) approach to the Goertzel algorithm is illustrated in Figure 5. The availability of local memories (M's) is not assumed here in order to explain the basic AP capabilities used. Shown in Figure 5 is an array of k words (k=10 for Digitone) by 6 fields; the field length is dependent on the accuracy required. Field F6 holds the k different frequency coefficients which remain unchanged throughout the iterative loop.

A new decompanded sample is written into field F1. Now the iterative computation can begin. F3 is moved to F5 then multiplied by F6 leaving the most significant bits (msb) in F4. F2 is subtracted from F4 and F1 is added to F4. After shifting fields F1 to F4 to the left by the field length, we are ready for the next sample. At the end of N samples, F4 and F5 hold the power of the frequencies.

Bit Fields

	F1	F2	F3	F4	F5	F6
1	X(nT)	Y(nT-2T)	Y(nT-T)			C_1
2	X(nT)	Y(nT-2T)	Y(nT-T)			C_2
.	.	.	.			.
.	.	.	.			.
.	.	.	.			.
k	X(nT)	Y(nT-2T)	Y(nT-T)			C_k
	Current sample	Next to last output	Last output	Work Space		Frequency coeffi-cients

Fig.5. Large AP Array Layout for the Comutation of a Discrete Fourier Transform by the Goertzel Algorithm.

B. Compact Approach

From the operation just described, it seems clear that not all the fields take part in the manipulation at the same time; e.g., during the multiplication of F3 and F6, F1 and F2 need not be present. In fact, only fields F4 and F5 need to have an arithmetic capability if fast bit-to-bit communication is available to the other fields. This in essence is the rationale behind the BNR associative processor cell architecture which calls for eight M pages (local memories) and one A page (auxiliary memory or travelling accumulators). It will be demonstrated in this paper that a 12 by 12 array is capable of performing this computation for two channels by storing the operands in the M's and doing calculations in the A. Accuracy longer than the word length can be achieved by factoring the operands into short pieces then combining their partial results.

Proper placement of operands in the M's is crucial to the optimum operation of the array, and nowhere is it more evident than in the placement of frequency coefficients and the windowing factor. Figure 6 shows the 10 frequency coefficients (for Digitone) broken up into three equal parts, stored in bits 0-3 of words 2-11 in pages M0-M2. They remain there until the receiver switches to a different set of frequencies (e.g., MF). The 8-bit windowing factor is stored in bits 4-11 of word 0 and repeated in word 1 of page M4. A new factor must be loaded for each new sample. The undesignated memories will be allocated later.

C. The Basic 4 by 8 Multiplication

Basic to the iterative computation is the 4 by 8 bits multiplication shown in Figure 7. It is a successive, conditional addition algorithm commonly found in many computers except done in parallel for any number of words. For simplicity only two words are shown here. One of the M's holds the multiplicands in bits 4-11. The A is initially loaded with the multipliers in bits 0-3. By loading one bit-slice at a time from the multipliers into the mask register (MR), the partial sums which need to be added are selected. Steps (1) to (6) show the changes of A as the partial sums are built up to the final 12-bit product.

D. The 16 by 12 Multiplication

The most complex operation in the Goertzel iterative equation is the multiplication of y(nT-T) by Cw as indicated in Figure 4. This is a 12 by 16 bits multiplication giving a possible 28-bit product from which the most significant 16 bits are kept. Figure 8 shows the use of the basic 4 by 8 multiplication to achieve this result. The 16-bit multiplicand is expressed as the sum of two 8-bit terms, and similarly the 12-bit multiplier is expressed as the sum of three 4-bit terms. The sum of the six partial products after scaling constitutes the final product. However, since only the 16 most significant bits are needed, some simplifications are possible:

a. Step (1) may be ignored because it does not contribute to the 16 msb.
b. In step (2), only the 4 msb from the product is saved in the work space.
c. After step (3) is done, the result from (2) is added and only the 8 msb are saved.
d. The result of (3) is added to the result of (4) and the 8 msb are saved.
e. All of step (5) is saved.
f. The result of step (6) is right-shifted 4 places and then the result of (5) is added. This is the final result.

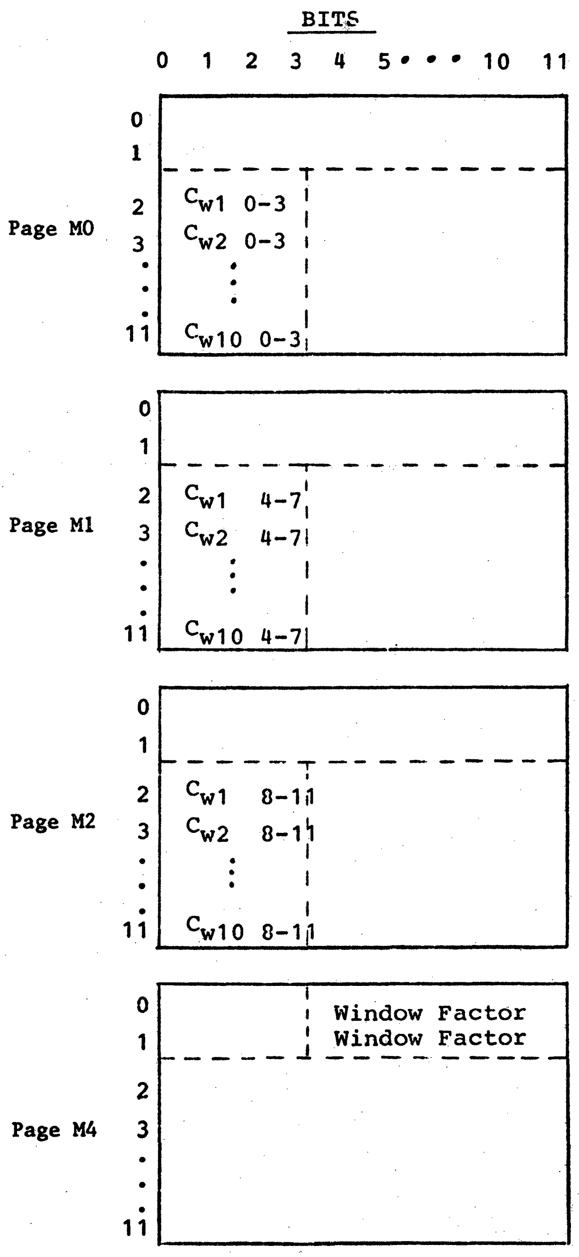


Fig.6. Loading of Goertzel Frequency Coeffients and Window Factor

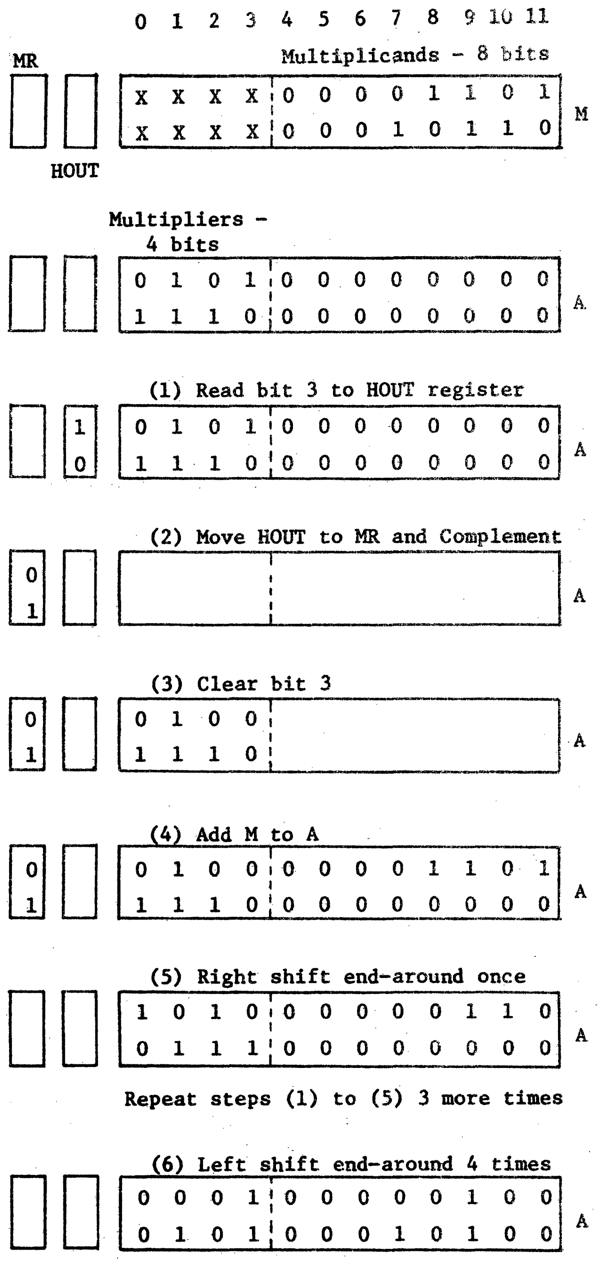


Fig.7. 4-bit by 8-bit Multiplications

$$(\text{Multiplicand 16 bits}) * (\text{Multiplier 12 bits})$$
$$= (\text{8 upper} + \text{8 lower}) * (\text{4 upper} + \text{4 med.} + \text{4 lower})$$
$$= (Ud.2^{8} + Ld) * (Ur.2^{8} + Mr.2^{4} + Lr)$$
$$= Ud.Ur.2^{16} + Ud.Mr.2^{12} + Ud.Lr.2^{8} + Ld.Ur.2^{8} + Ld.Mr.2^{4} + Ld.Lr$$

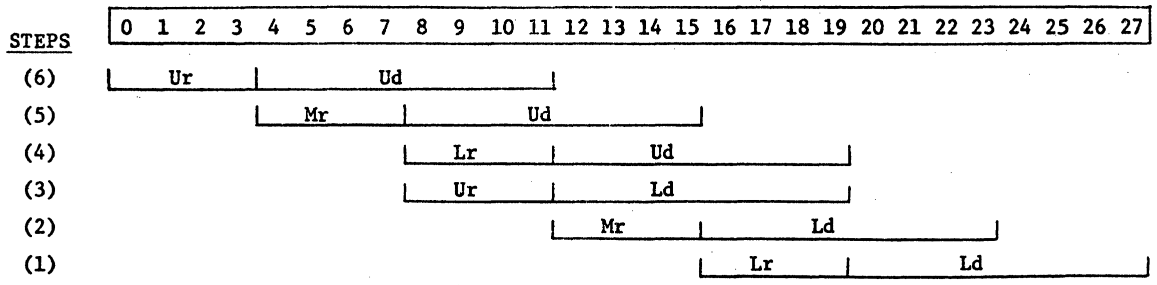


Fig. 8. The 16-bit by 12-bit Multiplication

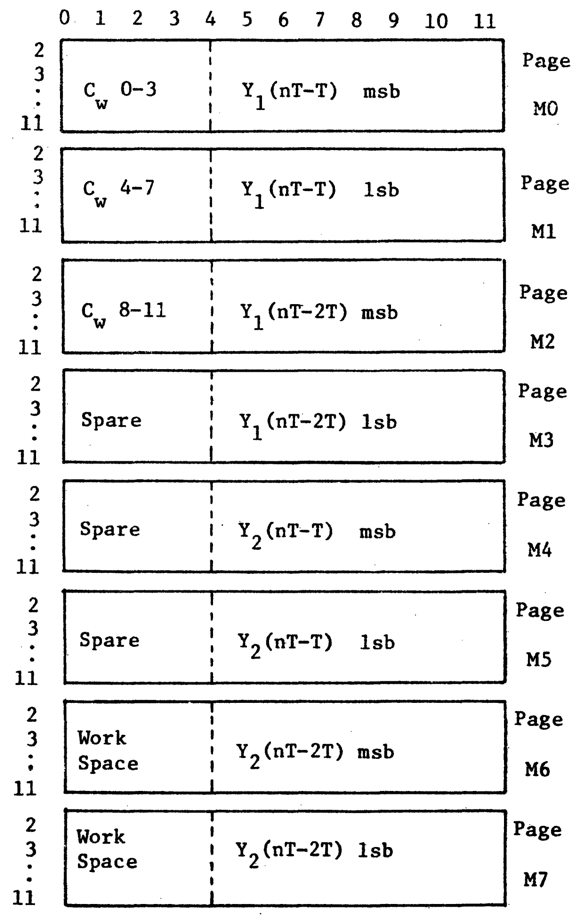


Figure 9: The Allocation of Operands for the Iterative Computation of two channels

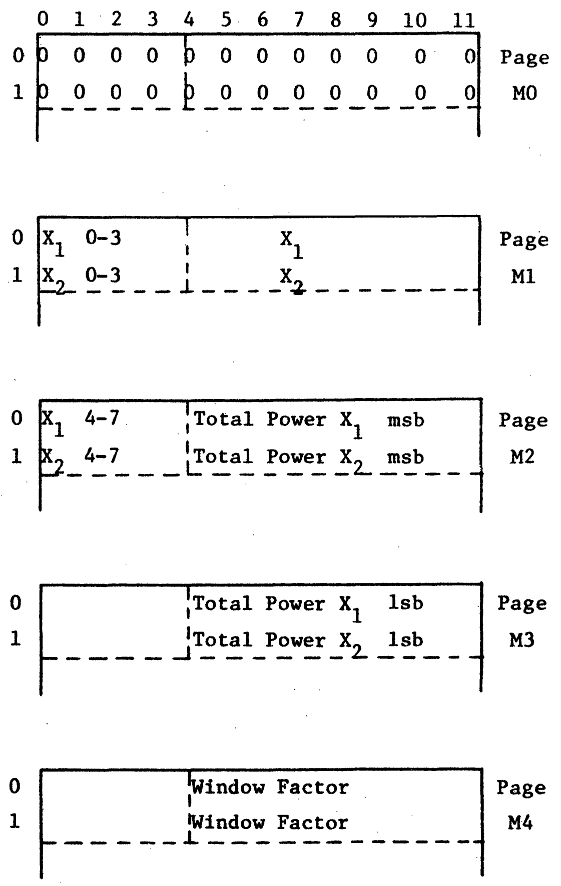


Figure 10: Squaring and Total Power Accumulation

E. Iterative Computation

In order to perform this piecemeal multiplication efficiently, the sub-operands must be placed for easy access. This is the reason why the frequency coefficients are so distributed. Figure 9 shows the allocation of the sub-multiplicands and working spaces. Channel 1 (x1) uses pages M0 to M3 in which the 16 bits of y(nT-T) occupy M0 and M1 while y(nT-2T) occupies M2 and M3. Similarly channel 2 (x2)uses pages M4 to M7. Thus the two samples are operated on by the same procedure one at a time using the same sets of multipliers and work spaces.

After the Cw and y(nT-T) are multiplied, the product is split up between the work spaces and the A. Subtraction of y(nT-2T) is done by first operating on the least significant bits (lsb), adding the carry (from the HOUT register) to the msb of y(nT-2T) then operate on the most significant bits. The difference is stored in the position of y(nT-2T). After adding the x(nT) to the least significant bits the data are ready for shuffling in preparation for the next iteration. This is simply swapping between the y(nT-2T) positions and the y(nT-T) positions.

The last part of the iterative computation is the total power accumulation. The squaring operation is an 8 by 8 multiplication which can be split up similar to the Cw and y(nT-T) multiplication. By placing the operands in the manner shown in Figure 10, it can be performed at the same time as Cw.y(nT-T).

F. Simulation

Using an AP simulation program in IBM 370/168, we have simulated the 4 by 8 multiplication. Execution speed has been estimated to be about 7.05 microsec by means of this simulation and preliminary call design parameters.

G. Execution Time

As stated earlier, the receiver design stands or falls on its ability to complete the iterative computation within the sampling period of 125 microsec. Since it's a dual receiver, the time available for the computation is cut to half or 62.5 microsec. The following shows the calculated execution time for each part:

Window multiplication	9.35 microsec
Decompanding	2.62
Cw.y(nT-T)	41.98
+x(nT)	1.20
-y(nT-2T)	2.20
Total Time	57.35 microsec

This is within the sampling period.

7. CONCLUSION

This paper has shown the technical feasibility of a receiver design using custom designed associative processing cells. Assuming a 12 by 12 array, it has demonstrated that such a receiver is capable of handling two channels producing 16-bit accuracy and still have time and memory spaces to spare. However, it is a flexible design because both the accuracy and execution speed can be improved by increasing the word length and/or the number of words per channel.

References

[1] Robert H. Beeman, 'Improved Multifrequency Receiver for 2/6 Code Communications Switching', IEEE Transaction on Communication Technology, vol. COM-18, No. 3, pp 165-167, June 1970.

[2] Kunihiko Niwa, Mitsutake Sato, 'Multifrequency receiver for Pushbutton Signalling Using Digital Processing Techniques', IEEE International Conference on Communications, pp 18f-1-5, June 1974.

[3] Ivan Koval, George Gara, 'Digital MF Receiver Using Discrete Fourier Transform', IEEE Transaction on Communication, vol. COM-21, No. 12, pp 1331-1335, December 1973.

[4] Jackylene Hood, Maitang Mark, Jack Cotton, 'Architecture and Simulation of an Associative Processor Integrated Circuit' International Conference on Parallel Processing, Wayne State University, August 1976

[5] Member of the Technical Staff, Bell Telephone Laboratories, 'Transmission Systems for Communication', 4th edition, Bell Telephone Laboratories, Inc.

[6] B. Gold, C.M. Rader, 'Digital Processing of Signals', McGraw-Hill, 1969.

RADAR DATA PROCESSING ON THE ALAP

Hubert H. Love, Jr.
Strategic Systems Division
Hughes Aircraft Company
Los Angeles, California 90005

Abstract -- The distinguishing features and some typical operations are described for the Associative Linear Array Processor (ALAP), a highly-parallel computer. Emphasis is given to operations, employing a chaining channel, that permit parallel arithmetic to be performed between the contents of sets of cells. Next, an application program for the ALAP, which performs radar track correlation, association and prediction, is described, and several programming techniques are illustrated.

Introduction

The Associative Linear Array Processor is the result of an internally funded development effort at Hughes Aircraft, the objective of which was the development of a low-cost associative memory suitable for both arithmetic and non-arithmetic applications. The project effort has resulted in the design and fabrication of a complete associative processor system, including LSI wafers containing the ALAP cells, and the programming of support software and application programs for the system. The subject of this paper is one of the application programs, which performs several of the more critical functions of radar data processing. The program has been written and checked out, using a symbolic assembler and a simulator program operating on the Sigma 9 computer.

The remaining sections of the paper give a very brief description of the ALAP design, a more thorough discussion of the general programming techniques for the ALAP for arithmetic applications, and a description of the radar data processing program itself. Emphasis is given to the chaining channel, which is one of the more unusual features of the ALAP, in both the hardware description and the programming techniques discussion. A more thorough description of the ALAP design is found in reference [1]. Brief descriptions of the ALAP design and a non-arithmetic application are given in references [2] and [3], respectively.

General Description of the ALAP

Figure 1 shows the general organization of the ALAP Demonstrator System. The principal component of the system is the ALAP memory module, in which essentially all computation, except for some I/O processing, is performed. Programs and constants reside in the general-purpose processor, a General Data NOVA minicomputer. The minicomputer controls the sequencing of the instructions and furnishes the control information and data for the instructions to the ALAP memory through the Interface Unit. The Interface Unit can hold several instructions in advance of their execution, and thus permits the ALAP memory to operate with minimal delay between instruction cycles. The ALAP Demonstrator Unit interfaces with the user through a Teletype.

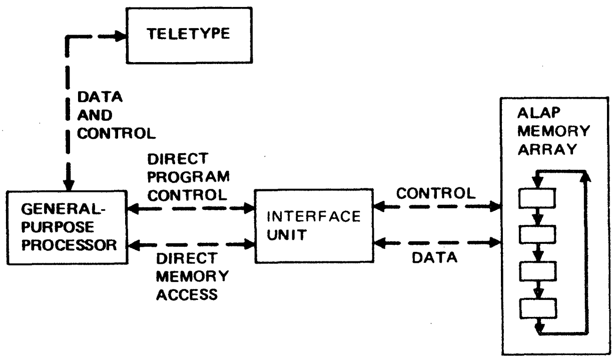


Figure 1. The ALAP Demonstrator

The general organization of the ALAP memory is illustrated in Figure 2. The ALAP memory consists of an arbitrary number of associative cells interfacing with four communication channels. Two of these channels are common busses that permit common items of data to be input from the Interface Unit to one or more (software-selected) ALAP cells simultaneously. The third channel, also a common buss, permits data to be output from one or more (software-selected) ALAP cells to the Interface Unit. (If data from more than one cell is output, the data is logically OR-ed on the channel.)

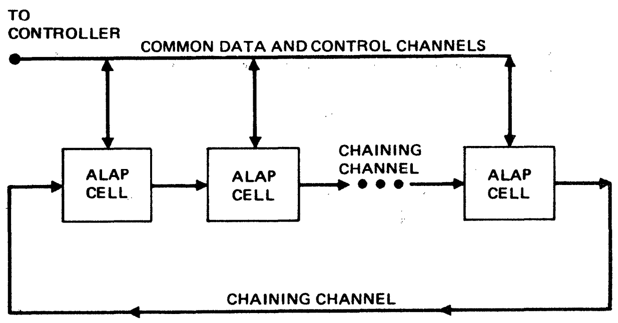


Figure 2. The ALAP Memory Array General Organization

The fourth channel, the "chaining channel", connects each cell to its neighbor, and thus organizes the cells into a linear array. This channel is not a common buss. The data output onto this channel from each cell during program execution will in general be different at different cells. The chaining channel transfers data in one direction only, a fact that, as will be described, leads to some interesting programming techniques. The chaining channel and all three of the common channels are bit-serial in operation.

The figure shows the array connected "end-around" with respect to the chaining channel. This is a software-controlled option; the first and last cells can be linked or not as desired for the particular operation being performed.

An ALAP memory wafer contains all chaining channel logic for its cells, as well as the remaining cell logic. An entire wafer has only 20 external connections; this number is independent of the number of cells which the wafer contains.

Figure 3 is a simplified diagram of the structure of an ALAP cell. The cell holds its data in a bit-serial "data register". In the ALAP memory wafers fabricated at Hughes, the data registers are 64 bits in length. The data register interfaces with the cell's arithmetic logic, the chaining channel and the three common channels (the latter are not illustrated by the diagram) by means of logic that is set under program control at the individual cells. The state of this logic for each cell is determined by the settings of the bits in the cell's "flag register". This is a six-bit register, also bit-serial in operation, that interfaces with the communication channels. A separate flag, called the "head flag", together with some additional logic, permits the flag settings in the flag register to be rearranged as the register is shifted, and to be AND-ed and OR-ed together in the process, if desired.

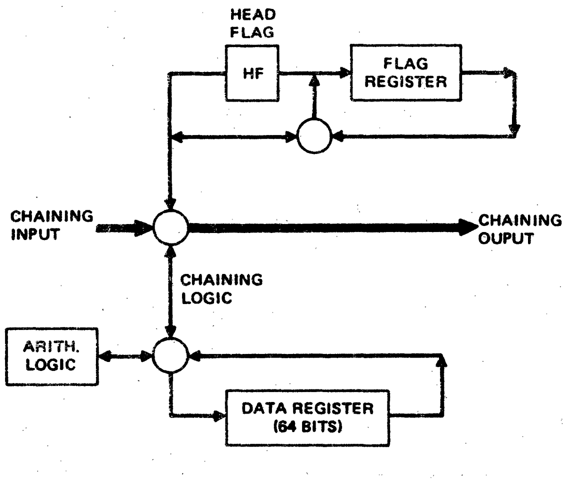


Figure 3. The ALAP Cell General Organization

The principal instruction-execution operation performed by the ALAP memory is called the "word-cycle" operation. This operation consists of shifting the data registers of a selected subset of the cells. The number of bit positions shifted is software controlled. It is usually 64, the length of the data registers. During this operation, data coming into the cell from the chaining channel or one of the common channels may replace the contents of the data register, or else may be arithmetically or logically combined with the contents of the data register. The results of an arithmetic operation at the cell may replace the contents of the cell's data register, if desired. At the same time, either the previous contents of the data register or the results of the arithmetic operation may be output onto the chaining channel to the next cell. Alternatively, during a word-cycle operation, the cell may simply act as a relay for the chaining channel data, transferring the incoming data onto the next cell in the array and performing no other function.

The arithmetic or logical operation performed at a cell during a word-cycle operation is determined by the settings of global control lines common to all cells. The selection of the cells at which the operation takes place is made by the setting of one of the bits in the flag register. In the cells thus selected, the operations take place between the incoming data and the data register contents. They include exact match, addition, subtraction, step-multiplication and step-division.

The operation of the chaining channel logic in each cell during a word-cycle operation is determined principally by the settings of two of the bits in the cell's flag register. With respect to the chaining channel, then, the cells can be considered for practical purposes to operate independently of one another. The settings of other flag bits in a cell during word-cycle operations variously determine whether data is to be input to or output from the cell via one of the common channels, whether a match operation was successful, or whether overflow occurred during an arithmetic operation.

In addition to the word-cycle operation with its various options, there is a class of subordinate operations, called "flag-shift" operations, that are performed in the ALAP memory. These operations consist of shifting the flag registers at all cells while performing logical operations at each cell among the register contents, the head flag and the input from the chaining channel (the latter consisting of flag information from the previous cell in the array.) The states of selected flags may be output via the chaining channel to the next cell in the array during flag shift operations.

The general operation of the ALAP memory during program execution consists of alternating sequences of flag-shift operations, which set the states of the flag register bits and head flags as desired, followed by single word-cycle operations during which each cell performs according to the combination of global control states and its internal flag settings.

Figure 4 illustrates the way in which the cells in a segment of an ALAP memory perform parallel arithmetic operations, using their chaining and arithmetic logic. These cells are set up to calculate two separate sums, A+B+C+D and F+G, during a single word-cycle operation. The first cell in the segment contains the operand A. During the word-cycle operation, this cell shifts the contents of its data register onto its chaining channel output to the second cell. The second cell contains the operand B. During the word-cycle operation, this cell adds the data at its chaining channel input to the contents of its data register, shifting the sum onto its chaining channel output. The chaining logic in the third and fourth cells are set to relay state; these cells relay the chaining channel data to the fifth cell. At the fifth cell, another addition takes place, and the sum is relayed past the sixth cell to the seventh. Here the final sum, A+B+C+D, is calculated. However, instead of being output to the next cell, it is stored in the same cell, replacing the cell's previous contents, D. Simultaneously with the computation of this sum, the other sum, F+G, is calculated from the contents of the eighth and ninth cells and stored in the last cell. This entire operation is bit-serial; both partial sums are calculated for each bit of the operands in turn as all of the data registers are shifted. One clock cycle time is required for each bit. Since there are several gate delays at each cell because of chaining and arithmetic logic, the clock rate, which is program-controlled, is set slow enough so that each bit of the initial operands A and F can propagate through the entire sequence of operations before the next bit is processed.

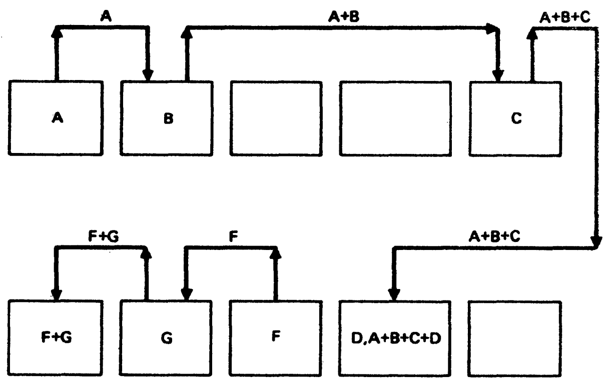


Figure 4. Arithmetic Operations Using the Chaining Channel

General Programming Techniques

The ALAP design is general-purpose in that it is suitable both for arithmetic applications, such as the radar data processing application to be described, and for such non-arithmetic applications as fact retrieval and text processing. In the latter two applications, the ALAP memory is often programmed to operate as a single long shift register, using the chaining channel. This alleviates many problems normally encountered in processing variable-length data items in fixed-word-length machines.

In arithmetic applications, the programming techniques for the ALAP are quite different. The radar data processing program in particular represents an example of what might be termed a "block-oriented" application with respect to the parallel-processing techniques which are employed. That is, for this application, it is convenient to partition the ALAP memory (by software means) into "blocks" of contiguous cells, each containing the data and associated working storage (which must also be replicated if parallel processing is to be possible) for a single object being tracked.

Figure 5 illustrates the division of an ALAP memory into a number of blocks. The memory is connected "end-around" with respect to the chaining channel, the purpose for which will later be apparent.

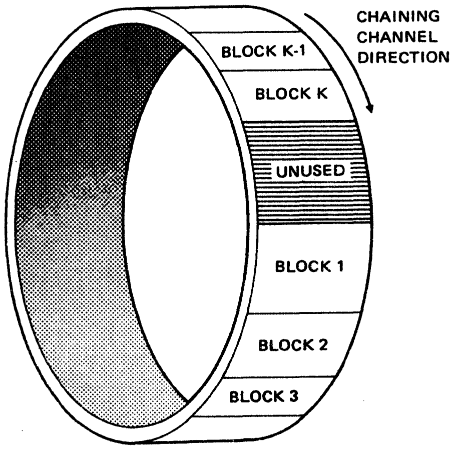


Figure 5. General Memory Layout for Block-Oriented Data Processing

The technique for processing all blocks in parallel requires that the corresponding operands and working storage be in the same relative cell locations within all blocks. In addition, the first cell in each block is reserved as a "header word". All header words have a special tag in a reserved field, thus enabling them to be identified and tagged by means of a single parallel match operation. Once this is accomplished, flag shift operations can both set the head flags at all cells to the states of their corresponding match flags and then, using the chaining channel, can advance all of the head flag settings past any desired number of cells simultaneously.

The result of this sequence of operations is to leave the head flags set at exactly those cells in all blocks that are to be set to a particular arithmetic and chaining state. Subsequent flag shift operations can then logically OR the head flag states at all cells with the states of the desired corresponding flag register bits. The OR operation ensures that the flag settings will be made at only those cells in which the head flag is initially set (to 1).

Figure 6 shows a block of 69 ALAP cells as they are employed in evaluating a set of six arithmetic functions. The set of functions is

WORD NO.	INITIAL CONTENTS	AFTER COPIES	AFTER FIRST MULTIPLY	AFTER SECOND MULTIPLY	AFTER ADDITION	AFTER SUBTRACTION
1.	Header	Header	Header	Header	Header	Header
2.	D(4)					
3.	P(1,1)					
4.			D(4) P(1,1)			
5.	P(1,2)					
6.			D(4) P(1,2)			
7.	2					
8.				2 D(4) P(1,2)		
9.	P(1,3)					
10.			D(4), P(1,3)			
11.				2 D(4) P(1,3)		
12.	P(1,2)					
13.				2 P(1,2)		
14.	P(1,3)					
15.				2 P(1,3)		
16.	4					
17.	P(2,2)					
18.				4 P(2,2)		
19.	8					
20.	P(2,3)					
21.		4		8 P(2,3)		
22.	P(3,3)					
23.			4 P(3,3)		P'(1,1)	
24.		D(4)				
25.		P(1,2)				
26.			D(4) P(1,2)			
27.		P(1,3)				
28.			D(4) P(1,3)			
29.		2				
30.				2 D(4) P(1,3)		
31.		P(2,2)				
32.			2 P(2,2)			
33.	6					
34.		P(2,3)				
35.			6 P(2,3)			
36.		4				
37.		P(3,3)				
38.			4 P(3,3)		P'(1,2)	
39.		D(4)				
40.		P(1,3)				
41.			D(4) P(1,3)			
42.		2				
43.		P(2,3)				
44.			2 P(2,3)			
45.		P(3,3)				
46.			2 P(3,3)		P'(1,3)	
47.	D(5)					
48.		P(1,2)				
49.			D(5) P(1,2)			
50.		P(1,3)				
51.			D(5) P(1,3)			
52.		2				
53.				2 D(5) P(1,3)		
54.		P(2,2)				
55.		4				
56.		P(2,3)				
57.				4 P(2,3)		
58.		P(3,3)				
59.				4 P(3,3)	P' (2,2)	
60.		P(1,3)				
61.			D(5) P(1,3)			
62.		P(2,3)				
63.		2				
64.		P(3,3)				
65.			2 P(3,3)		P'(2,3)	
66.		P(3,3)				
67.	D(3)					
68.		D(3) P(1,3)				
69.	K				P (3,3) + K	P' (1,3)

Figure 6. Memory Layout for Arithmetic Example

taken from the track prediction part of the radar data processing program. This figure will help in describing the way in which arithmetic operations are combined within blocks, as well as being performed for all blocks simultaneously. It will also be used as an example to illustrate the flag-setup processes described in the preceding paragraphs.

The set of functions to be evaluated is the following:

$$
\begin{aligned}
P'(1,1) &= D(4)P(1,1) + 2D(4)P(1,2) \\
&\quad + 2D(4)P(1,3) + 2P(1,2) + 2P(1,3) \\
&\quad + 4P(2,2) + 8P(2,3) + 4P(3,3) \\
P'(1,2) &= D(4)P(1,2) + 2D(4)P(1,3) + 2P(2,2) \\
&\quad + 6P(2,3) + 4P(3,3) \\
P'(1,3) &= D(4)P(1,3) + 2P(2,3) + 2P(3,3) \\
P'(2,2) &= D(5)P(1,2) + 2D(5)P(1,3) + P(2,2) \\
&\quad + 4P(2,3) + 4P(3,3) \\
P'(2,3) &= D(5)P(1,3) + P(2,3) + 2P(3,3) \\
P'(3,3) &= P(3,3) - D(3)P(1,3) + K
\end{aligned}
$$

The evaluation of these functions, if performed by a serial processor, requires 20 additions, 1 subtraction and 28 multiplications (including shift operations for multiplying by powers of 2 in fixed point). By combining operations within the block, the ALAP memory can perform the evaluation with a total of one addition, one subtraction, two multiplications and 9 copy operations, independent of the number of blocks. (The copy operations, necessary in rearranging data within the block, have approximately the same execution time as additions with the same number of operands and the same relative spacings between cells.)

Figure 6 contains seven columns of figures. The first of these is a list of the word numbers for the ALAP cells whose contents are illustrated. (These numbers are assigned by the programmer only for convenience, since the hardware is sensitive only to cell order or relative position.) The direction of the chaining channel is in the order of increasing cell number. The second column contains the initial cell contents before initiation of calculations. The third column shows the changes in the cell contents resulting from the nine copy operations. The remaining columns show the changes in cell contents for each of the remaining operations.

The initial operation in evaluating the six functions is that of setting all ALAP cells to the relay chaining state. This is the default chaining state for all arithmetic operations. Next, the match flags are set at all cells and a parallel match operation is made to identify all header words. The match flags will be reset at all nonmatching cells by this operation. Next, flag shift operations are performed to set the head flags at all words to the states of their corresponding match flags.

The process of setting up the cells to perform the copy operations now begins. A flag shift operation is executed which transfers the state of the head flag at each cell via the chaining channel to the head flag at the next cell. This operation requires one clock time.

With the head flags now set at Cell 2 of all blocks, flag shift operations using OR and AND functions are used to set the flag registers at these cells so that they will shift their data register contents onto the chaining channel during the next word cycle operation. The operation of chaining the head flag states is now repeated for 22 clock times, thus setting the head flags at Cell 24 of every block. These cells are then set to input the data from the chaining channel into their data registers during the next word cycle operation. The data will also be output on the chaining channel from these cells, thus making a multiple copy operation possible.

Next, the head flag chaining operation is repeated for 15 clock times, and all Cell 39's are set to the same state as the Cell 24's. The set-up operation for the first copy operation is now complete, and a word-cycle operation is next executed, copying the operand D(4) from Cell 2 of every block into Cell 24 and Cell 39, with all intervening words being in the default relay state as previously set.

The remaining copy operations shown in the third column of the figure are performed in similar fashion to the first, except that some time can usually be saved by having previously stored the states of the head flags at the header words in some otherwise unused flags. This avoids having to repeat the match operation each time; the flag-shift operations needed to reset the head flags are much faster.

The first of the two multiply operations is next set up and executed. Flag shift operations similar to those for the copy operations are used to set the flag registers in the cells containing the multipliers, multiplicands and products to corresponding chaining and arithmetic states. It will be noticed that a single multiplier can take part in more than one multiplication operation simultaneously as long as there are no conflicts on the chaining channel. For example, the operand D(4) in Cell 2 is multiplied by the operands in Cells 3, 5, and 9 to produce products in Cells 4, 6, and 10, respectively.

Multiplication is performed by a subroutine which performs repeated step-multiply operations The number of step-multiply operations is equal to the number of bits in the multiplier. As each step-multiply operation requires a complete word cycle, the multiplication process is the most time-consuming operation of the entire arithmetic calculation process.

The second multiplication operation follows the same order of setup and execution as the first. The single addition operation is performed next. In this operation, several operands are added together at once. For example, all of Cells 4, 8, 11, 13, 15, 18, 21 and 23 are added together, with the sum being put in Cell 23.

After the addition operation is completed, all of the functions except the last have been evaluated. The subtraction operation is then used to complete the evaluation of the last function. The cell states for subtraction are the same as for addition, but the word cycle operation includes the global subtraction command rather than addition. In each of the columns in the figure the cell contents are shown only for those cells whose contents have changed since the preceding operation.

Some interesting problems arise in block-oriented applications because of the uni-directional chaining channel. The radar data processing program is cyclic. It repeats all of its operations during each cycle, operating on new data extracted from the input from the radar antenna and on data calculated during the previous cycle. In particular, the values of the six functions evaluated in the example just discussed will be used in evaluating these same functions during the next cycle. For example, the value of P'(1,1) will be the next value of P (1,1). It is apparent that the chaining channel presents a problem when the data in the block must be rearranged to put it in the original relative order prior to reevaluating the functions. For example, P'(1,3) cannot be moved back from cell 69 to cells 9 and 14 to be the new P(1,3).

Since the data cannot be moved backward within the blocks along the chaining channel, the solution to the problem is to move the blocks forward instead. That is, instead of moving the new function values back, the program moves everything else in the blocks forward, including the header words, thus effectively relocating all of the blocks. Since this is done parallel by block for each item, the time penalty is not as great as it might first appear.

One can indeed ask why the ALAP cell was not originally designed with a two-way chaining channel, in order that the data reordering problems, and other difficulties associated with the one-way chaining channel might more easily be handled. The answer is that low-cost producibility is a principal objective of the ALAP design. In the applications thus far explored, the benefits to be gained in ease of programming and lower execution time do not appear to be justified by the cost of the additional cell logic. This new logic must result in a greater cell area on the LSI wafers, and thus in fewer cells per wafer. This means a higher proportional cost per ALAP cell, no matter how inexpensive LSI fabrication may become.

The Radar Data Processing Program

The radar data processing program performs the three functions of track correlation, association and prediction on track data from a radar antenna. This data has been preprocessed to remove obvious redundancies and to eliminate tracks which clearly do not represent objects of interest (e.g., the strength of the return or the apparent velocity of the object are outside of nominal limits imposed by the application).

The data for the apparent tracks (called "observations") which remain are the inputs to the ALAP program. This data consists of five parameters for each observation: the range, range rate-of-change and the three direction cosines. The program operates in cycles. Each cycle consists of performing the three aforementioned tracking functions in turn on new sets of observations.

Track correlation is the process of comparing the five track parameters of each observation in turn with the corresponding parameters for all tracks being maintained by the program. (These are tracks for known "targets" or "threats", depending on whether the radar is for an offensive or defensive application.) If all five parameters for an observation fall within a preassigned range of the corresponding parameters for a known track, the observation is considered to have "correlated" to the known track, and thus may represent an update of that track.

In the ALAP program, all five parameters of a single observation are compared with all five parameters of all stored tracks simultaneously. The five parameters are then input, one at a time, into all blocks simultaneously at which correlation has been successful. The blocks must all be large enough to contain the parameters for several correlating observations. Correlation is very efficiently performed by the ALAP program. A total of 22 +24S word cycle operations is required, where S is the number of observations. If there are 200 observations, and if the ALAP clock rate is 5 MHz., the elapsed time for correlation is 8.3 msec, including a 30 percent overhead for flag-shift operations.

After the correlation process has finished, there will in general be cases in which a single observation has correlated to more than one stored track, and in which many observations have correlated to the same stored track. Track association is the process of removing both of these types of redundancy.

The ALAP program performs well at this function, though not so efficiently as for correlation. The program first computes an error function for each correlated observation in each track. This function is

$$\text{ERROR} = \left(\frac{r_o - r_p}{\sigma_r}\right)^2 + \left(\frac{\dot{r}_o - \dot{r}_p}{\sigma_{\dot{r}}}\right)^2 + \left(\frac{\lambda N_o - N_p}{\sigma_{\lambda N}}\right)^2 + \left(\frac{\lambda E_o - \lambda E_p}{\sigma_{\lambda E}}\right)^2 + \left(\frac{\lambda D_o - \lambda D_p}{\sigma_{\lambda D}}\right)^2 .$$

The five terms of this function are for the range, range rate and the three direction cosines, respectively. The subscript o indicates that the parameter is for an observation. The subscript p indicates that it is for a stored ("predicted") track. The denominators in the fractions are constants in the program.

The computation of this error term for all correlations requires a total of 146 word cycles (1.9 msec with a 5 mHz clock). The single multiplication and the single division each account for 65 of these.

The program next resolves all cases in which a single observation correlated to more than one track. This is done serially by observation, and consists of deleting the parameters for the observation from all tracks except the one having the smallest error term for that observation. This requires a total of 9B +5C word cycles, where B is the number of observations and C is the total number of correlations for all of the observations together. If there are 200 observations, and if 100 of them each correlate to two stored tracks, the elapsed time is 46.5 msec for a 5 mHz clock, including a 30 percent overhead for flag shift operations.

Last, the association part of the program resolves the cases in which more than one observation correlated to a single stored track. This is done by removing from the corresponding block the parameters for all observations except that having the smallest error term. This is done in parallel for all blocks, and requires a total of 41 word cycles (equivalent to 6.8 msec for a 5 mHz clock, including 30 percent flag shift overhead).

Track prediction is the third and last of the radar processing operations. This consists of evaluating 29 functions, of which the six functions in the example of the previous section are the first. The 29 functions, if evaluated on a serial computer, require a total of 51 additions, 17 subtractions, 47 multiplications and 4 divisions for each stored track. By combining the operations in the fashion described in the example of the previous section, the ALAP program performs the evaluations parallel by track with a total of 8 additions, 1 subtraction, 4 multiplications and 2 divisions. Some of the functions are dependent on others, else only two multiplications, one division and a single addition and subtraction would have been required altogether. To this total must be added 41 multiple copy operations (needed for reordering the data for the next cycle) and 24 parallel match operations, making a total of 475 word cycle operations altogether. This requires 7.9 msec with a 5 mHz clock.

Of particular overall significance in the radar data processing program is the fact that transfers of data between the Interface Unit and the ALAP memory are limited to initial input of the observation data and output, if desired, of the updated track data at the conclusion of each correlation/association/prediction cycle. At only one point in the program is it necessary to transfer intermediate information to the minicomputer from the ALAP. This is a single word which is checked for a 1 or 0 in order to select a program path. Thus the program appears to demonstrate one requirement for efficient use of associative processing techniques, namely that there be a minimum demand on the relatively slow serial input and output capability of the processor as compared with the parallel processing capability.

References

[1] B.F. Meyers, The Hughes Associative Processor, Computer Science Dept., University of California at Los Angeles, Modeling and Measurement note No. 26, May 1974.

[2] C.A. Finnila, The Associative Linear Array Processor, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, Syracuse University.

[3] H.H. Love, Programming the Associative Linear Array Processor, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, Syracuse University.

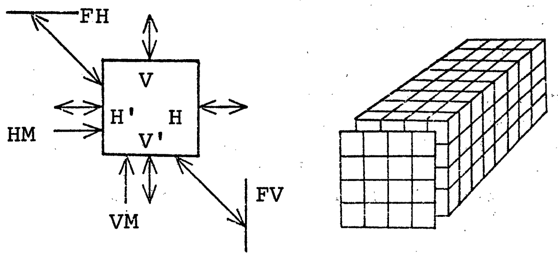
ARCHITECTURE AND SIMULATION OF AN ASSOCIATIVE PROCESSOR INTEGRATED CIRCUIT

Jackylene Hood, Maitang Mark, and Jack Cotton[(a)]
Bell-Northern Research
Ottawa, Ontario, Canada

Summary

The associative processor cell is a storage and processing element capable of performing Boolean and arithmetic operations. Similar circuits were designed at University College of London [1] and at Stanford Research Institute [2].

The operations in an associative processor array require a direct mask line, a direct data line/wire-ANDed search results line, and a ripple shift/carry line in each direction.



For an array instruction such as Add Field A to Field B the addends must be brought together. The A's could be shifted over the intervening bits but only one bit slice at a time. If the entire Field A were shifted simultaneously, the intervening data would be destroyed. So a second storage element, termed a travelling accumulator, is added to each cell. The main storage element is an 8-bit shift register so an array of 4 x 4 cells may be considered as 9 pages of 16 cells each. The front page is the set of travelling accumulators.

The microcommands can be divided into seven groups: logic, arithmetic, search, zero test, input/output, and general. The auxiliary and main memory elements can be logically combined, added, or subtracted. An external constant can be added to multiple accumulators simultaneously. Equality searches use the wire-ANDed lines, but inequality searches (less than, less than or equal, greater than, greater than or equal) must use ripple lines and are consequently slower. Ten extra commands are provided by not clocking the main commands. For example, an unclocked OR command tells the system if an OR Main to Accumulator instruction would give a zero result.

The shift commands use FV and FH as control lines so that 20 shifts are provided by only 8 microcommands. Included are logic shifts (pad with zero), arithmetic shifts (sign extend), rotations, jump shifts and propagation shifts. All shifts are effective in four directions.

(a) Now with ITT TTC, Stamford, Connecticut

The main and auxiliary memory elements may be read simultaneously to both column and row output registers. Horizontal and vertical masks provide the addressing. A rotate command moves all cells to the next circular shift register position.

The APSIM, a user interactive software package operating on an IBM 370/168 was developed to simulate the manipulation of arrays of AP cells and registers. The interactive feature enables users to evaluate simulation results and to modify them instantly via a terminal input mode. A series of commands can be processed via a disk input mode. Both modes may be intermixed within one simulation.

The APSIM employs a modular approach that anticipates frequent changes in command functions. Thus an independent program module is used for each micro-command, so that a change or an addition of command can be accommodated simply by altering or adding one module. Uniform logic structure of each program module is strictly enforced to aid the maintainability of APSIM. Each program module has access to a common data base where registers, auxiliary cells, and up to eight levels of 256 x 256 AP memory cells are stored. The simulation also accepts simulation instructions generating loops, branches and subroutines.

In addition to the work reported here, this project includes the hardware and software design of a general purpose real-time associative computer. Circuit design and board layouts have been completed for three black box applications of this chip: Multifrequency/Digitone reception [3], television bandwidth compression, and speech bandwidth reduction. Prototype chips will be fabricated late this year.

References

[1] M.J.B. Duff et al., "A Cellular Logic Array for Image Processing," Pattern Recognition (vol. 5, 1973), pp. 229-247.

[2] W.L. Kautz, "An Augmented Content-Addressed Memory Array for Implementation with Large-Scale Integration," Journal of the Association for Computing Machinery (January, 1971), pp. 19-33.

[3] E. Shew and J. Cotton, "A Receiver for PCM Coded Digitone and MF Signals Using Associative Processing," Wayne State International Conference on Parallel Processing (August, 1976).

APPLICATION OF PEPE TO REAL-TIME DIGITAL FILTERING

D.B. Kimsey, L.E. Hand, and H.T. Nagle, Jr.
Electrical Engineering Department
Auburn University
Auburn, Alabama, 36830

SUMMARY

This paper describes the application of the IC model of PEPE [1] to real-time digital filtering. The PEPE IC model resides in a research laboratory at Auburn University configured with a dual mini-computer host. The host computers are a Honeywell H316 and a Hewlett-Packard 2100. PEPE consists of sixteen parallel units, each with a 32-bit Arithmetic Unit (AU), a 512-word, 32-bit data memory, and an 8-cell, 40-bit correlation unit. The mini-computer hosts send global commands to the PEPE Arithmetic Control Unit (ACU) which has no program memory of its own. The hosts may also transmit programs to the PEPE Correlation Control Unit which has a 512-word, 32-bit program memory.

Since the AU's possess a full complement of fixed and floating-point instructions, one can implement a 32nd-order digital filter as a parallel of 16 second-order modules, each module being implemented in one PEPE AU. Floating-point arithmetic was used in this experiment alleviating what would have been a time-consuming allignment problem. The input from the host's A/D was simply inserted into the high-order fraction bits creating a zero-exponent, (generally un-normalized), floating-point number. The output of the filter was un-normalized to a zero exponent and the high-order fraction bits shipped to the host's D/A.

The frequency sampling technique of digital filter design [2] lends itself very well to an implementation on a parallel processor. The filter structure begins with the comb filter $1-z^{-N}$ (implemented in the host) which places N zeroes on the unit circle. The pass band is then formed by cancelling up to 32 of these zeroes by a sum of 1st-order poles in parallel; each of the form

$$G_k = \frac{H_k}{N} \frac{1}{1-z^{-1}e^{j2\pi k/N}}$$

where H_k is the desired magnitude of the response at that particular frequency.

Complex conjugate poles are paired and the H_k's forced to meet $H_k=H_{N-k}$ yielding the total transfer function

$$G = (1-z^{-N})\left[\frac{H_0}{N}\left(\frac{1}{1-z^{-1}}\right) + \frac{H_{N/2}}{N}\left(\frac{1}{1+z^{-1}}\right) + \sum_{k=1}^{(N-2)/2} (-1)^k \frac{2H_k}{N}\left(\frac{1-z^{-1}\cos(2\pi k/N)}{1-z^{-1}2\cos(2\pi k/N) + z^{-2}}\right)\right]$$

The H_0 and $H_{N/2}$ terms are combined in one AU and the remaining AU's each contain one of the 2nd-order modules in the summation. Computations for these 2nd-order modules are performed in parallel; however, the summing of the outputs of these modules becomes a serial problem compounded by the lack of inter-processor communication in PEPE. Approximately 380 μsec of the total 600 μsec for one sample period is due to this serial summing; thus a single "sum all accummulators" instruction could almost triple the present maximum sample rate of 1670 Hz. The order of this filter is unlimited as long as the number of non-zero H_k's does not exceed 32.

A slightly higher sample rate may be obtained using a cascaded implementation where each AU contains a second-order module whose output is the input to the next second-order module. The modules are computed in parallel, but outputs then have to be shifted to inputs (to the next AU) in a serial fashion. This is essentially a pipelining process. The cascaded structure thus obtained introduces a delay of z^{-15} but the overall thruput is improved over that of the parallel structure since serial shifting of outputs is faster than serial summing.

An N^{th}-order Butterworth low-pass filter having unity DC gain and an upper cutoff of one radian is given by the transfer function

$$D(z) = \prod_{k=0}^{N/2-1}\left(\frac{1 - 2e^{-\alpha_k T}\cos(\beta_k T) + e^{-2\alpha_k T}}{1 - 2e^{-\alpha_k T}\cos(\beta_k T)z^{-1} + e^{-2\alpha_k T}z^{-2}}\right)$$

where α_k = SIN $(2k+1)\pi/(2N)$
β_k = COS $(2k+1)\pi/(2N)$
T = sample period in seconds.

A 32nd-order Butterworth low-pass filter was impelmented in PEPE using a 2nd-order module in each AU. The execution time for one sample period was 500 μsec of which 290 was used for the serial propagation of outputs. Thus a single "propagate accumulators right" instruction could more than double the present maximum sample rate of 2KHz.

Several transfer functions having the above two structures were programmed and the frequency response curves were plotted experimentally. Stability was excellent despite the high order of the filters, and all curves agreed with the theoretical curves.

REFERENCES

[1] D.B. Kimsey, "A Fast Fourier Transform Program for the PEPE IC Model", Proceedings of the 8th Annual Southeastern Symposium on System Theory, (April, 1976), pp. 153-155.

[2] C.M. Rader and B. Gold, "Digital Filter Design Techniques in the Frequency Domain", Proc. IEEE, (Feb., 1967), pp. 149-171.

HIGH LEVEL LANGUAGE FOR ASSOCIATIVE AND PARALLEL COMPUTATION WITH STARAN [a]

R. G. Lange
Digital Technology Department
Goodyear Aerospace Corp.
Akron, Ohio 44315

Abstract - The design and manufacture of computer systems with parallel and vector processing capabilities have brought about much activity in the area of higher level languages. Reports of efforts to design languages with parallel processing and parallel operation features have been made (Ref. 1). An effort is in progress to design and fully specify a higher level language for STARAN [b]. The specification is scheduled for completion this year.

Introduction

We first state the objectives and requirements for the language and then show the structure of a program and the form for declarations. Next, we describe parallel and associative operations and a sample procedure, an image processing algorithm for following lines.

Language Objectives and Requirements

The higher level language for STARAN is a procedural programming language. It is designed for programmers to use to implement algorithms that accomplish an application function. Thus, it does not attempt to be a specialized problem oriented language. The language definition emphasizes parallel operations on data items, provides for declaration of the data items, and provides expressions and many operations for the use of data items. The language is statement oriented with statements for assignment and other operations.

The design objectives for the higher level programming language are:

1. The language must be reasonably complete functionally in order to allow the major portion of a problem solution to be written in it. The language must provide array and search (associative) operations.

2. The language must support the program design process including structured programming and other methodologies of software engineering.

3. The language design should have good human-factors characteristics; the language is a human-machine interface.

4. The language must be implementable in order to generate STARAN machine language code and make good use of its architectural properties.

5. The language must be kept small but yet large enough to meet the above objectives.

Data declarations are provided for specification of the range of values of data items, thus supporting conservation of execution time and storage space and also providing documentation of the valid data range. They provide for precision for arithmetic items, length of bit and character oriented data, and size of arrays.

To aid in the support of structured design, the language emphasizes abstract data representation (machine independent) and structuring of data items. Multiple entry procedures (routine statements) allow data to be localized to one procedure and thus improve cohesion within the procedure.

The data organizations provided are simple (scalar) data items, simple structures, structures of arrays (serial arrays), and arrays of structures (parallel arrays). These data facilities and others allow the programmer to group related data items that also may require similar allocation within a memory type or area.

In order that most of an application can be written in the language, it is relatively rich in the number and meaning of operations specified. The language design attempts to minimize the number of special rules, such as context-dependent exceptional cases.

Programs written in this language should be as readable as possible to reduce costs in both checkout and maintenance. The language supports the use of meaningful names for program and data entities; names may be up to 32 characters in length with all characters significant. The tokens of the low-level syntax are designed in a manner to allow indentation of the source program (listing), optionally by compilers, according to the program's structure.

The development of structured computer programs is finding increased acceptance within the computing community because of indications that the code written in this manner is more

[a] This project was partially supported under RADC Contract F30602-76-C-0200.

[b] Trademark, Goodyear Aerospace Corporation, Akron, Ohio 44315.

readable, understandable, maintainable, and reliable at lower cost. The language supports structured programming and structured design through its various data declaration and proceduring facilities; it emphasizes the program flow aspects of structured programming with an appropriate set of flow of control statements.

Program Structure

The language is statement-oriented in form (versus expression-oriented as in ALGOL); all statements other than the assignment statement begin with an introductory keyword such as IF or CALL. In order to enhance program clarity and to avoid ambiguity, some of these keywords are reserved. The order and types of statements used in a program determine the flow of execution for that program.

A program consists of one or more <program unit>s together with their operating environment. A <program unit> is the largest syntactic construct of the language and serves as the unit of input to the compiler.

The set of <program unit>s that constitute a <program> is determined by CALL statements and function references during execution of the program or by use of the linking facility prior to execution.

The syntax of a <program unit> is just an <external procedure>.

A <block> is an entire <external procedure> or any <procedure> contained in another <procedure>. It delimits the scope of name declaration and is the major unit that determines program flow of control during execution.

Syntax:

```
<program> ::= <program unit>
<program unit> ::= <external procedure>
<block> ::= <procedure>
<external procedure> ::= <procedure>
<procedure> ::= <procedure statement>
                <procedure body>
                <endproc statement>
<procedure body> ::=
          <procedure component>
           [<routine statement>
             <procedure component>] ...
<procedure component> ::= <statement>...

<statement> ::= <procedure>
               |<declare statement>
               |<basic statement>
       |<prefix><basic statement>
<basic statement> ::= <group>
       |<independent statement>
<independent statement> ::=
         <single statement>
       |<conditional statement>
```

```
<prefix> ::= <label prefix>
          |<case prefix>
<label prefix> ::= <identifier> :
<case prefix> ::= CASE ( <case number> ):
<case number> ::= <cvi expression>
   |<cvi expression> : <cvi expression>
```

A <routine statement> defines additional entry points for a procedure as exclusive sequences of statements; a return is implied for the previous entry point.

A <group> is a construct used to determine the flow of control during program execution. Groups are of two types: loops and case selection.

Syntax:

```
  <group> ::= <iterative loop>
             |<repetitive loop>
             |<case group>
```

Individual statements are classified as declarative statements or procedural statements. There is one declarative statement, the <declare statement> for declaring data. The procedural statements are used to form the executable statements of a <program unit>. Some of them represent individual statements; however, the <if statement>s must include a corresponding <endif>. The individual statements are called <single statement>s.

Syntax:

```
<single statement> ::=
          <assignment statement>
        | <call statement>
        | <close statement>
        | <delete statement>
        | <goto statement>
        | <null statement>
        | <open statement>
        | <read statement>
        | <return statement>
        | <rewrite statement>
        | <write statement>
```

All statements are executable, although the execution of a <declare statement> or <null statement> has no effect. The close, delete, open, read, rewrite, and write statements are "record I/O" statements and are not discussed further in this paper.

The ";" symbol is used to delimit statements; it is a statement terminator and is shown with the statements.

Statement Prefixes

A <label prefix> is a means of naming a <statement>. In certain contexts, such as within a loop, any <basic statement> may be named by being preceded by a <label prefix>.

Declarations

The STARAN language has strong typing of data, as do most other procedure-oriented programming languages. There are additional data properties expressed via data attributes in order to characterize the data length or precision, its allocation and life-time requirements, and its structure or relationship to other data items.

These properties are due in part to the need to adequately describe the use of data that will be allocated within STARAN array memories. The attributes and placement of the data declarations in the source program establish the scope of the data.

Data Types

Five types of computational data are defined; in addition, there are data types "file" and "entry." Each of the former types is different in internal representation and in the values it may assume. The specification of data types is concerned with the abstract properties of the data rather than the internal representation. Thus, the storage requirements for each type are not specified.

Computational data is further separated into arithmetic and string types. The arithmetic data types are cardinal (attribute is CARDINAL), fixed-point (attribute is FIXED), and floating-point (attribute is FLOAT). The string data types are bit-string (attribute is BIT) and character-string (attribute is CHARACTER).

Arithmetic data is specified with a precision to indicate the number of bits necessary to represent its values; for FIXED, this may also specify a scale factor for fractional values. Arithmetic data is represented in a binary base, and string data is a contiguous sequence of bits or characters.

A cardinal data item represents an unsigned integral value stored as a binary number; it may assume only zero or positive integral values.

Data Organization

To meet the total requirements placed upon the language, the data items can be organized as arrays or structures or they can be individual data items.

A scalar is a single element of data or one member of a set of data elements. A scalar may appear in a program as a constant or as a variable representing one element of data.

Arrays. An array is an ordered set of scalars, all having identical attributes; it is identified by a single symbolic name. An array appears in a program as a variable representing a set of scalars.

The unique identification of an array element consists of the array name and the position of the element in the array. The position in the array is indicated by a bracketed subscript list following the name; for example:

```
image_table [line_number+1]
```

The elements of an array are stored as an ordered sequence so that the rightmost subscript varies most rapidly and the leftmost subscript varies least rapidly. This order is required by the interaction between arrays and structures and is called row-major order.

An array is declared by appending a dimension specification to the name in a DECLARE statement.

```
DECLARE image_table (256) CARDINAL (8);
```

Structures. A structure is an ordered set of data elements which may have different attributes. The elements may be scalars or arrays. A relationship exists between data elements of a structure. The relationship is indicated by level numbers in the declaration. The main structure is a level-one; nested structures are given numbers greater than one to indicate their logical level.

The main structure, nested structures, and the innermost data elements all have names. Qualification is used to uniquely identify a nested structure or data element name. A qualified name consists of the name of the main structure and all nested structures leading to the structure or data element name to be identified, with a period operator as the separator between each pair of names.

Arrays of Structures. An array of structures is a structure with the attribute dimension following the level-one name in the declaration. Each element of the array is thus a copy of a structure with structuring identical to all other array elements. An example is:

```
DECLARE 1 record (64),
          2 field1 CHARACTER (12),
          2 key      CARDINAL (8),
          2 properties BIT (4),
          2 index    CARDINAL (16);
```

Other Attributes

The language has other data attributes such as for the control of storage class: automatic, controlled, and static. There is an external attribute to allow linking of data between separately compiled programs.

The memory attribute allows the program to force allocation of an array or array of structures in the special STARAN MDA (multi-dimension access) memory. There is also an alignment attribute that forces alignment to memory boundaries which are a power of two as requested.

Parallel Operations

As we have seen in the requirements, the language emphasizes parallel and associative operations. There are two motivations for this. One is that the language should support problem solution at a high level and allow index computation and other housekeeping operations to be done by the compiler. The other is to allow parallel operations of the target machine STARAN to be used in the object program.

The parallel operation emphasis is primarily in four areas. First, there is the extension of the familiar expression operators to arrays and structures as operands. Second, there is the inclusion of a parallel if test for use with vectors. Third, there is the addition of a new notation for selecting only a portion of an array's elements in a reference to the array. The fourth is the inclusion of several built-in functions for arrays and the ability to have user functions reference any array (of a fixed dimension).

Extended Operators

First of all, the infix and prefix operators that may appear in expressions can have scalar or structure operands and the operation given is performed on all items in parallel. The operations include add, subtract, multiply, divide, relational comparisons, concatenation, logical-AND, logical-OR, and logical complement.

For example, if A and B are 128 element arrays of fixed-point data, then

```
A < B
```

is a 128-element array of single-bit strings. We usually call the latter a bit-vector. It can be assigned to a variable of that type or tested in an IFARRAY statement to control other computations.

IFARRAY Statement

The IFARRAY statement has the form:

```
IFARRAY <vector valued expression>
    THEN
        <basic statement> ...
   [ELSE
        <basic statement> ...]
ENDIF;
```

A simple example shows its use:

```
DECLARE ( A(200), B(200) ) FIXED(15, 0);
DECLARE COUNT (200) FIXED (15);

IFARRAY A < B THEN
 A [&] = B;
ELSE
 A [&] = A * 2;
 COUNT [&] = COUNT + 1;
ENDIF;
```

For each element of A that is less than the corresponding element of B, assign the latter to replace the value of A. In all other cases, double the value of the element of A and increment a counter for each such element. The ampersand used as a subscript (in context of IFARRAY-THEN) means select the subset of only those array elements corresponding to B'1' values in the nearest containing IFARRAY <expression>. Ampersand used as a subscript in the context of the ELSE clause means select the subset of only those array elements corresponding to B'0' values in the IFARRAY <expression>.

Array Selection Subscripts

There are other forms of array-selection that provide parallel usage. Similar to the above use of ampersand alone is

```
<array name>  [& <expression>]
```

where <expression> is an array matching <array name> and has values convertible to data type BIT(1). It selects those elements of <array name> corresponding to elements in <expression> having value B'1'.

An asterisk in a subscript position selects all elements accessible according to use of all valid subscripts in that subscript position. This is a cross-section reference as in PL/I. Another form of cross-section reference defined is with subscripts of the form:

```
<lower limit> : <upper limit>,
```

where both of these are <expression>s and their values during execution select a portion of the array elements in combination with the values of other subscripts, if any.

Built-in Functions

Many built-in functions are extended in a manner similar to that done for expression operators so that their operands can be arrays or structures. Examples are SIN, COS, SQRT, SUBSTR, and INDEX. Others have been added to make the parallel facilities more complete. INDEX_MIN provides the index of (one of) the minimum values of an array rather than the minimum value itself. This is useful in searches where unique values are required. INTERVAL_TEST is a convenient function for range testing. For example:

```
IFARRAY INTERVAL_TEST ( x, prev.x, delta)
   THEN ---
```

is the same as:

```
IFARRAY ABS( x - prev.x ) <= delta
  THEN ---
```

Applications

The appendix shows a sample procedure written in the language. It is a portion of an image-processing program. Even though it does not exhibit much use of parallelism, it makes special use of some other facilities for which STARAN is especially well suited. These facilities are primarily the row and column cross-sections (bit slices) of the array "data" and the single-bit tests. The procedure is about 14 percent of the number of lines of the assembly language version and is far more readable and maintainable.

There are other portions of the image processing application such as line thinning or clutter elimination that use much parallelism, executing two orders of magnitude faster than on serial processors.

Summary

The language as designed for the set of requirements as stated above is quite large, although it initially was proposed as smaller. This is primarily due to more emphasis on objective one versus the other objectives mentioned early in the paper, especially objective five.

Although there was some interest in having a language similar in facility to FORTRAN, because of the nature of some of the requirements the final language is more like PL/I, although still a small subset (dialect) of it. The facilities required, when considered as a whole, were beyond the scope of FORTRAN. Many of the basic concepts that have a counterpart in FORTRAN (such as expression evaluation, assignment, and serial loop control) remain similar to FORTRAN and its current standardization (ANSI) proposal.

Reference

1. Proceedings of the Conference on Programming Languages and Compilers for Parallel and Vector Machines, ACM, March 1975.

Appendix - Line Following

One of the reasons for choosing this algorithm as a candidate for coding is the nature of the process performed. The data being processed is essentially binary ones and zeroes representing the raster scanned data of a map. It is observed that the degree of effort and clarity of representation of the program written in high level language are good tests for the programming language.

The line following procedure vectorizes all lines in an array and outputs the data to various tables. A previous routine has performed (in parallel) such functions as line thinning (reducing a line to a single cell in width), clutter elimination, and tagging all lines in the array by storing the starting coordinates in a table. A starting coordinate may be a boundary point or an interior point. Beginning with a starting point, the procedure follows the points along a line segment.

Definition of Variable Names

DATA is a two-dimensional array of single-bit items and contains the points of the lines to be vectorized. It is oversized by one cell on each border so that all starting points will be "interior points." In this way, no check need be made when following a line to ensure that the dimensions of DATA are not exceeded.

START.TAG is an array of single-bit items containing a one-bit value in each index position of a starting coordinate; that is, values in the arrays START.X and START.Y are selected. START.X and START.Y are arrays of (x, y) coordinate starting point values of the various lines contained in the DATA array.

PREV_DIRECTION is a variable containing the direction of the previous successful search along the line. NEXT_DIRECTION is an array of values containing the next point along the line and is a function of PREV_DIRECTION.

TABLE_OF_TRIES is an array containing the maximum number of possible directions in which the succeeding point along a line may be found.

NUMBER_OF_POINTS is a variable containing the number of points (or vectors) discovered in a line.

FOUND_SUCCESSOR is a single-bit tag indicating whether the line following procedure was successful at detecting a succeeding point. A value of one means another point was found; a value of zero means the end of a line has been detected.

DIR is a do-loop index and represents the range of directions to be tried.

CUR_DIR is the direction currently being tried and is a value between 0 and 7. Directions are numbered as:

```
5  6  7
4  .  0
3  2  1
```

VECTOR_LIST is a two-dimensional array containing the results of the line following procedure. It has a capacity for 32 individual lines, each containing up to 256 vector values (up to 256 points).

VECTOR_NUMBER_POINTS is an array containing the number of points for each of the 32 possible lines contained in VECTOR_LIST.

Line Following Procedure

```
follow_lines:  PROCEDURE;

  DECLARE  data(194, 194) BIT(1) MEMORY(mda),
           prev_direction CARDINAL(4),
           number_of_directions CARDINAL(4);

  DECLARE  index CARDINAL(5),
           1 start(32) MEMORY(mda),
              2 tag        BIT(1),
              2 x        CARDINAL(16),
              2 y        CARDINAL(16),
           vector_number_points(32) CARDINAL(16),
           vector_list(32, 256) CARDINAL(4);

  DECLARE  found_successor BIT(1),
           number_of_points CARDINAL(16),
           (dir, cur_dir) CARDINAL(4),
           (i, j) CARDINAL(8) ;

     DECLARE  table_of_tries(8) CARDINAL(4) CONSTANT
                  INITIAL (3, 5, 3, 5, 3, 5, 3, 5),
              next_direction(8) CARDINAL(4) CONSTANT
                  INITIAL(7, 7, 1, 1, 3, 3, 5, 5) ;

  /* Clear border around DATA area.  */

  data [1, *] = B'0';                   /* top     */
  data [DIM(data, 1), *] = B'0';        /* bottom  */
  data [*, 1] = B'0';                   /* left    */
  data [*, DIM(data, 2)] = B'0';        /* right   */

  /* Examine bit vector start.tag for ones;
     continue to loop until all start points
     have been processed.
  */
  index = 0 ;
  DO WHILE( SOME(start.tag) ) ;

     index = index + 1 ;
     start [index].tag = 0 ;

     /* Start looking in direction 1; try up to
        7 possible directions as necessary.
     */
     prev_direction = 1 ;
     number_of_directions = 7 ;

     /* Obtain coordinates of start point. */

     i = start [index] .x ;
     j = start [index] .y ;
     number_of_points = 0;

     /* Assume another point will be found.  */

     found_successor = B'1' ;
     DO WHILE( found_successor ) ;

        /* Assume end point will be found */

        found_successor = B'0' ;
  line_follow:
        DO dir = prev_direction TO
            prev_direction + number_of_directions ;
            cur_dir = MOD( dir-1, 8 ) ;
     /* Check direction cur_dir for a 1 bit */
        DO CASE cur_dir ;

CASE(0):          IF(data[i, j+1] )
                       found successor = B'1' ;
CASE(1):          IF(data[i+1, j+1] )
                       found successor = B'1' ;
CASE(2):          IF(data[i+1, j] )
                       found successor = B'1' ;
CASE(3):          IF(data[i+1, j-1] )
                       found successor = B'1' ;
CASE(4):          IF(data[i, j-1] )
                       found successor = B'1' ;
CASE(5):          IF(data[i-1, j-1] )
                       found successor = B'1' ;
CASE(6):          IF(data[i-1, j] )
                       found successor = B'1' ;
CASE(7):          IF(data[i-1, j+1] )
                       found successor = B'1' ;

         ENDCASE ;
         IF (found_successor)
            EXIT line_follow;
      ENDDO line_follow ;

      IF found_successor THEN
         /* Enter direction discovered. */

         number_of_points = number_of_points + 1;
         vector_list [index, number_of_points] =
                cur_dir ;

         /* Start search for next point based
            on direction of current point.
         */
         prev_direction = next_direction [cur_dir+1] ;

         /* Number of possible directions is
            a function of current point direction.
         */
         number_of_directions =
                table_of_tries [cur_dir+1 ] ;
      ENDIF ;
   ENDDO ;
   /* An end point was found; store total
      number of points for current line.
   */
   vector_number_points [index ] = number_of_points;
ENDDO ;
ENDPROC follow_lines ;
```

ANALYSIS OF THE AWACS PASSIVE TRACKING ALGORITHMS ON THE RADCAP STARAN

By Robert Katz

Boeing Computer Services, Inc.
Space and Military Applications Division
P.O. Box 24346
Seattle, Washington 98124

Abstract -- This paper analyzes the computer performance of the Passive Tracking programs which are operational on the RADCAP STARAN as well as an IBM 360/65. A brief review of the Passive Tracking functions is provided. Parallel program design considerations including first, second and third order parallelism, floating point software and data movements are detailed. Methods for both sequential and parallel computer performance measurements are discussed. Performance results in terms of timing and accuracy are presented. The comparison shows a timing advantage for the parallel version when the number of targets tracked exceeds 20. Conclusions regarding the reasons for this superior performance are given. The parallel performance is extended to consider potential timing advantages when tracking 1000 targets. Finally, recommendations useful for other RADCAP software applications are offered.

Introduction

Continuing research into parallel processing is being conducted at Rome Air Development Center (RADC) with the objective of assessing the efficacy of parallel processors to military applications distinguished both by high data rate requirements and by being beyond the scope of sequential processors. In particular, unclassified versions of Passive Tracking Algorithms based on the E-3A (AWACS - Airborne Warning and Control System) have been implemented on the RADCAP, (See Feldman et al [1]), the Goodyear Staran at RADC, as well as on a 360/65 at Boeing Computer Services, (BCS) in Seattle, Washington.

This paper presents an analysis of the computer performance in comparing the parallel (RADCAP) and sequential (360/65) program versions that were implemented. As part of this, timing methodology and results, as well as accuracies, are provided. It is an outgrowth of research (a) initially described by Prentice [5] and Prentice et al [6]. This paper also addresses the extent to which software design goals and concepts identified in Prentice [5] contributed to the superior parallel computer performance achieved. Conclusions, extrapolations and recommendations based on this parallel processing application study are offered.

(a) This research was performed by BCS for RADC under Air Force contracts F30602-74-C-0025 and F30602-75-C-0112.

Background

Passive Tracking is used to locate and track jamming targets from measurements of the radiation source azimuth. The Passive Tracking Algorithms maintain and update at each scan, a track history of targets. Three types of target tracking are treated by the program: self-passive (Mode 1), cooperative passive (Mode 0), and active (Mode 2). Each tracking mode differs in the number of AWACS aircraft receiving radar reports or scans as well as the information content of the radar scan.

In Self-Passive Tracking, a single AWACS aircraft flying in a closed loop receives radar returns containing target azimuth information only. In Cooperative Passive Tracking, this information is supplied by each of two AWACS aircraft. In this way the second aircraft's radar measurements are cross-told to those of the first aircraft. Finally, in Active Tracking, a single AWACS is used to capture the target azimuth and range information. Range information is available whenever a target, which operates its jamming equipment intermittently, stops jamming.

Input to the program for one radar scan involves radar azimuth and strobe width data (or range data) for each observed target along with the AWACS position(s). Output from the program consists of the calculated position, velocity, and quality for each target track. Figure 1 shows the flow chart for the Passive Tracking Program. In the sequential FORTRAN program, each underlined function performed by passive tracking requires an interior loop to process each track. In the parallel program, the array memory allows all track related variables to be processed simultaneously.

For each radar scan, six functions are used for the tracking activity:

- Track prediction
- Window computation
- Association
- Correlation
- Smoothing
- Deghosting

Track prediction involves determining the location and velocity of each target for the current scan based on that for the previous scan. A Kalman filter, used in track smoothing, is similarly predicted. The window computation determines the width of the window to be used in correlation based on the track prediction information. The association function produces a one-to-one correspondence between the set of computed, predicted target azimuths and the input target azimuth set such that the component differences are minimized.

Correlation tests if each input azimuth with its strobe width falls wholly or partially within the computed window of the associated track. The smoothing function modifies and updates the track location and velocity information based on the predicted Kalman filter, the predicted track information, and the correlated target azimuths. When resmoothing during active tracking, correlated target ranges are used. When resmoothing cooperatively, the updating additionally involves a 2-scan time delay in processing the secondary AWACS cross-told information. After this cooperative resmoothing, deghosting is performed to distinguish between n true targets and the remaining $(n^2 - n)$ strobe intersections (ghosts) via a track quality index compiled from range, velocity and acceleration limits applied to the smoothed tracks. Equations representing these functions are not shown here. They may be found in Prentice [5] or Lee [4].

Design of the parallel passive tracking program orchestrates all three RADCAP processors - the AP (Associative Processor), PIO (Parallel Input/Output Processor), and PDP/11 (Sequential Processor) - to execute 60 radar scans of information. All three processors can access bulk core and the high speed data buffer (HSDB) sequential memories. Also, the AP and PIO can access the associative array memory. For the program, the AP is the controlling processor and performs data movement, and computational activities. It also initiates sequential input/output requests and synchronization with the PIO via interlocks. The PIO is responsible for data movement and array (re)assignment when synchronized by the AP. The PDP/11 sequential processor is used to read and write data onto the disk from the high speed data buffer memory.

Figure 2 details the parallel version's program allocation and control interrelationships. Bulk core memory is used to store most program instructions and less frequently required variables. The page memory contains system subroutines as well as time critical program library subroutines. Included in these are both STARAN System and BCS floating point routines. The PIO Interpretive Code, located in bulk core memory, is processed by the code interpreter located in PIO memory. This interpreter enables a specific array reassignment or data transfer subroutine to move a set of data to, from, or within array memory. This processing is dependent on periodic AP synchronizations using the interlocks. The reader is referred to any of the last three references for additional background.

Design Considerations

There are three design aspects of the parallel version which substantially enhance its computer performance:

- o Parallelism
- o Floating Point Arithmetic
- o Data Movement

Although all three are concerned with program timing reductions, the floating point arithmetic used in the parallel version also enables numerical comparison and validation with the sequential program to be performed.

Parallelism

According to Prentice [5], the STARAN program was designed to exploit three types of parallelism: first, second, and third order parallelism. First order parallelism is concerned with the simultaneous operation on set components such as in Vector or Matrix addition. Second order parallelism deals with the movement, replication and alignment needed to perform simultaneous operations on each of the components of two or more diverse sets. For example, finding the product of a row and column matrix to form a square matrix requires all possible pairs of elements to be replicated, moved, and aligned prior to the matrix multiplication operation. Third order parallelism is the simultaneous operation of two or more computer processors such as the AP, PIO and PDP/11. It can be appreciated that suitable use of third order parallelism for manipulating data in advance allows second order parallelism to occur with the efficiency of first order parallelism.

To effect this situation, the AP and PIO synchronize during the passive tracking processing by means of interlock settings and sensings. The interlocks are flipflops and take on the functions shown in Table 1. The PIO shuttles, replicates, and aligns data among bulk core, the high speed data buffer and array memory for storage or for future use by the AP. Figure 3 shows the synchronization protocol for each processor. Note that the routines are functionally identical except for a PIO ready test by the AP. This is required to avoid a hangup when the previous AP operation is also a synchronization. After each return, the next respective AP and PIO operations are begun.

Floating Point Arithmetic

In order to numerically compare with the FORTRAN version which uses floating point arithmetic, the parallel version was written using Goodyear's (software implemented, two's complement) floating point package [2]. Note that the speed of a single precision floating point field to field multiply on the STARAN is 832 μ sec. This compares with 14 μ sec for that on the 360/65. In view of this, it became a design goal to minimize the number of floating point operations performed by the parallel program. In addition, a set of four additional single precision floating point routines were designed for parallelism, written in microcode, and tailored for maximum speed and minimum storage in the passive tracking program. These floating point routines are Cosine (Sine), Two Argument Arctangent, field negation or complementation (absolute value) and field to field comparison. Figure 4 shows the graphs of Cosine (I) and Arctangent (I1/I2) and their range of field input values.

The design philosophy of the trigonometric routines exploits the piecewise symmetry of their graphs. In each case, within the allowable range, the input field(s) is successively transformed and scaled to the basic approximation range. The polynomial approximation of the function (precise to seven significant decimal digits) is then computed for this scaled field. Finally, the output function is unraveled to correspond to its full range values. Each polynomial approximation is only performed once, since it is more time consuming (based on 5 multiplies and 3-4 adds) than the set of pre- and post-transformations. Table 2 describes the accuracies and timing and number of instructions required for these floating point routines. See Lee [4] for additional details regarding these routines.

Data Movement

Although the PIO provides for between-array data movement via a data driven subroutine, there are occasions when it is more efficient for the AP to provide this while maintaining the current array assignment. In particular, this occurs when the AP has no computations to perform; it can only wait while the PIO:

- o Completes synchronization
- o Reassigns arrays to PIO control
- o Transfers the data between arrays
- o Is resynchronized by the AP
- o Reassigns arrays back to AP control

The AP data movement routine was designed to operate via the Common Register and be comparable in speed. The comparison between the AP and PIO inter-array data movement routines (named ATADM and ATOA respectively) is pictured in Figure 5. Note that the AP routine transfers 32 bit words while the PIO routine transfers (up to) 256 bit slices. Table 3 provides measured and computed times for the AP and PIO inter-array data movements.

Measured times for the AP routine (ATADM) are taken for each tracking mode during the first two scans. The average time per call is based not only on the number of calls but the number of 32 bit words transferred per call. In the parallel program, this can range from 32 up to 256 words transferred per call. Computed times are shown to compare the AP routine with the PIO routine relative to the number of 32 bit words to transfer. In the PIO routine, vertical shifts are handled within a loop, thereby requiring additional time. In the table, the maximum shift, using a total of eight 32 word blocks (256 words), is based on the first source word to transfer being located at the top of the first word block, and the last destination word transferred being located at the bottom of the last word block.

Test Requirements

Both versions of passive tracking are to use the identical, simulated radar report input data. These data (on disk) contain random errors in azimuth values. These input data are compatible with both the IBM 360/65 and Honeywell 6180. (Once the data are on the HIS 6180 Multics System, they can be automatically converted and transmitted to the RADCAP disk). In order to validate the accuracy of the sequential and parallel versions, true target positions are provided for comparison. In analyzing the track history, the values at the end of the last (60th) scan for each version are compared to each other and with the true value. The criteria for validation is that the parallel computed values should differ from the sequential computed values by no more than 10% of the true values.

In both versions, the code was optimized to minimize the measured execution time. Furthermore, timing measurements exclusive of initialization and input/output activities were taken using hardware performance monitors. These measurement results are a function of the number of tracks, the number of radar scans, and the mode of tracking. Nine test cases, all based on 60 scans, were devised to explore these parameters. The first five test cases are concerned with the self-passive mode with the number of tracks varying from 64 to 16 in steps of 12. A single AWACS flying in a circle is assumed. The next three test cases explore the cooperative mode with the number of tracks varying from 32 to 16 in steps of 8. Two AWACS flying at opposite ends of a racetrack are assumed. The last case provides information on active mode of tracking using 64 tracked targets and a single AWACS. Figures 6 and 7 show the initial positions of the target set and the AWACS aircraft(s) for the nine test cases.

Test Methods

In the sequential version, the test cases were run three times to demonstrate repeatability. Three relocatable component load modules were provided for timing analysis. They were: execution control, the function of correlation and all other passive tracking functions. Each module was timed separately. The first run provided program analysis data. Run 2 provided repeatability, while run 3 with its load modules reversed, demonstrated proper hookup of the test instrumentation. CPU usage was measured using a Test Data System Utilization Monitor (SUM) and its associated 20 bit comparator [4]. The measurement errors of this equipment arise from the 1.0 microsecond resolution of the timing counters compared with the 360/65's 0.2 microsecond CPU cycle time and the non-interleaved memory cycle time of 0.75 microseconds. This influences the separately timed segments so as to always exceed the measured CPU problem state time.

In the parallel version, strict repeatability could not be provided. This was due to the independence of the PIO and PDP/11 processors combined with the dependence of the PIO on the AP and the PDP/11 on the AP. In particular, because of the variability of the PDP/11 disk seek times during untimed reading and writing, the subsequent "AP waiting periods for the PIO to be ready" which are timed, vary correspondingly. Figure 8 shows the situation.

The PIO has begun a synchronization process (including some data movement) which requires a fixed amount of time to complete. The

PDP/11 has begun an I/O process involving the disk that is completed in a variable amount of time. The AP waits if necessary for I/O completion. This wait time is not measured, but affects when the AP can issue the next synchronization to the PIO. If the synchronization is issued early, there is an increased and measured wait time for the PIO to be ready. If the synchronization is issued late, however, a decreased (perhaps zero) wait time is experienced. In this latter case, the PIO has received free processing time relative to the AP timing measurements, thus reducing the overall program timing. (To insure that no hardware malfunction was occurring and to support the concurrent processing contention, test case 6 (mode 0) was run twice for 60 scans with the PIO processor turned completely off. The results (per event) differed by only 0.132 microsecond, well within the known error tolerance of the performance monitor).

Thus, to provide indicative values, timing for each of the test cases was performed at least four times both through scan 2 and through scan 60. Subroutine timing for each tracking mode was also provided to assess the relative portions of time spent on each passive tracking function. Finally, total AP waiting periods for the PIO, for each mode, were separately timed to capture program delay time. All timing measurements were performed using the STARAN Performance Monitor (PFM) [3]. It consists of two 32 bit registers: an events counter, and a timer. The events counter measures the number of times that the timer is enabled, while the timer measures the execution times of one or more instructions. Up to 0.2 microsecond of error can arise due to timer resolution and enabling the timer.

Performance Results

In the sequential version, the correlation load module is a non-linear function of the average number of targets used in searching and number of tracks, whereas the other two load modules are linear functions of the number of tracks. Overall measured CPU problem state time and summed module component time by test case are given in Table 4. In all cases, the summed time exceeds the measured time.

For the parallel version, validation was performed by comparing computed sequential and true target positions. In an overwhelming number of instances, the maximum component difference in position per test case did not exceed 3/4 of a mile, representing less than 1% deviation. Active tracking, which utilizes additional range information, provided a maximum variation of 0.0033 of a mile. Table 5 lists the few variational anomalies that did occur. The four starred track numbers indicate significantly better positions for the parallel version. Each of the pairs in the last 3 columns represents positive component distances in miles.

Test case results show mode 1 (cases 1-5) to require an average of 3.9 seconds to execute 60 scans of data. Mode 0 (cases 6-8) requires nearly twice as much time whereas mode 2 only requires half again as much time. Test case timing is shown in Table 6. The average and extremal times through 3 scans and through 60 scans are provided. Also averages are computed representing the constant value for each mode. The calculated time derived by running component subroutines for each mode is additionally shown and is consistent with the average time for each mode.

The average AP waiting time for PIO is given in Table 7. Percentages are relative to average mode times of Table 6. Because of the relative constancy (40%) of this waiting time in all three modes, most of the waiting time appears to occur while the smoothing subroutine is processing. In fact, the successive synchronizations in this routine do support this contention. Too, the smoothing subroutine itself is the single most time consuming routine, accounting for 35% of the total mode 1 time. For mode 0 and mode 2, where this subroutine is called twice (for resmoothing) per scan, the percentages are even higher - 48% and 59% respectively.

The basic computer performance result comparing both versions is that the parallel version is superior to the sequential version:

- o For modes 1 and 2, at the 64 track maximum, it is more than 3 times faster
- o For mode 0, at the 32 track maximum, it is about $1\frac{1}{2}$ times faster

The overall timing comparison for the nine cases in both versions is provided in Figure 9. The crossover point is defined as the number of tracks handled by the sequential and parallel versions in the same time period. For the passive modes, the crossover occurs at approximately 20 tracks. Furthermore, in the active case, the parallel version is 3 times faster than the corresponding sequential version using 64 tracks.

Conclusions

Based on the reported comparison between parallel and sequential versions of Passive Tracking, the parallel version is at least as accurate as the sequential version. For 64 tracks in the self passive and active modes, the parallel version is more than three times faster; for 32 tracks in the cooperative mode, the parallel version is more than one and a half times as fast. The low 19 and 21 track crossover points in the passive modes are attributable to: 1) efficient, joint use of second and third order parallelism which adds to the inherent first order parallelism of the algorithms; (2) minimized number of floating point operations.

Finally, in analyzing the parallel program for timing bottlenecks, it appears that the floating point arithmetic routines account for 90-95% of the active AP processing time. It is conjectured that this may hold true for any STARAN application which uses the software floating point package.

Extrapolations

A further conclusion can be entertained about the trend of the results as the number of tracks goes beyond 64. Namely, the parallel program execution time in all modes increases more slowly in comparison to the increase in the number of tracks. From these parallel performance data, it is useful to consider what occurs for the passive tracking type of application when, for example, 1024 tracks are needed. What time savings are possible and what storage problems and program bottlenecks are there?

The parallel passive tracking program is designed for the 64 track case as a maximum. Thus, a step function (of size 64) describes the execution time (vs. the number of tracks) as the number of tracks increases to 1024. Furthermore, the step size depends on the degree to which the present design is frozen and memory (array memory especially) can be enlarged. This discussion presumes that a larger PDP/11 can be configured with 32K or 64K of sequential memory to provide for the increase in track dependent storage. The timing contributions of association and smoothing functions of the tracking are particularly sensitive since they currently utilize all existing array space.

Let vectors T_0, T_1, T_2, and T_3 be defined as follows:

T_0 = the total execution time (seconds) per mode using 64 tracks (from Table 6 average mode times) $= \begin{pmatrix} 3.9 \\ 7.2 \\ 5.5 \end{pmatrix}$

T_1 = the association function time per mode $= \begin{pmatrix} 0.9 \\ 1.5 \\ 0.8 \end{pmatrix}$

T_2 = the AP waiting for PIO time per mode (from Table 7) $= \begin{pmatrix} 1.5 \\ 2.7 \\ 2.3 \end{pmatrix}$

T_3 = the smoothing function time per mode $= \begin{pmatrix} 1.5 \\ 3.2 \\ 3.0 \end{pmatrix}$

In the first case, let the design be frozen and the array memory modules increased from 4 to 64. It is assumed that, as the number of tracks doubles, the AP inter-array data movement time portion of the association represents a little less than half (0.45) of T_1. Similarly, it is assumed that all program wait time for PIO data movement is concentrated in the smoothing routine (an overestimate). Thus, by successively doubling the number of arrays from 4 to 64, a factor of 4 is introduced. Symbolically, we get

$T_4 = T_0 + 4*(0.45*T_1 + T_2)$. Thus

$$T_4 = \begin{pmatrix} 7.9 \\ 14.2 \\ 10.6 \end{pmatrix}$$

which is an estimate of total execution time for the tracking of 1024 targets with a frozen design and 64 array memory modules. This represents a doubling of time relative to 64 tracks.

In the second case, let the array memory size be frozen (to 4 arrays) and let the design be modified so that computation times for association and smoothing are doubled (e.g. via multiple passes) each time the number of tracks is doubled. Let the entire data movement increase for both routines be represented by the increment $(T_3 - T_0)$, an over estimate. Then, by successively doubling the number of tracks from 64 to 1024, a fourfold power of two is introduced. Symbolically, this is

$T_5 = T_4 + 2^4 (T_1 + T_3)$. Thus

$$T_5 = \begin{pmatrix} 46.3 \\ 89.4 \\ 71.4 \end{pmatrix}$$

which is an estimate of total execution time for 1024 targets with 4 array memory modules and a modified design. This represents a 12-fold increase in time over that for 64 tracks.

Thus, it appears that increasing the number of tracks 16 times to 1024, increases program execution time by a factor of at most 12. This can be a significant time saving particularly when storage and design modifications are jointly undertaken.

Recommendations

It is recommended that for real time applications or applications in which minimized processing time is the objective, consideration should be given to the exploration of second and third order parallelism and the synchronization relationship between the two processors. To ease future program implementations on STARANs similar to and including the RADCAP, several software implementation recommendations are offered:

1. An extremely useful adjunct, while running the passive tracking program was the development by RADC of post-processing floating point conversion routines on Multics. Two routines were used, one to convert unformatted binary to decimal values and the other to convert formatted (hexadecimal) ASCII to decimal values. The main feature of these routines was the speed at which many thousands of numbers could be converted. Since no conversion software exists on STARAN currently, it is recommended that these routines be made available within Goodyear system software as an addition to the post processing capabilities of SDM.

2. Portions of the program were written in pure microcode while other portions were heavily APPLE assembler language oriented. Certainly the microcode, once checked out, is cleaner, faster, more compact code. However, if minimizing program checkout time is the main objective, then assembler language should be utilized throughout. On the other hand, if minimizing program storage and timing is the objective, microcoding of key subroutines and processes should be exploited from the onset. It is recommended that consideration be given to the proper mix of language types

used in application programs.

3. A rather disciplined approach to the coding and checkout of the passive tracking program was undertaken. At the microcode level, symbolic register status maps were heavily used. These maps provided an updatable code line correspondence with the symbolic variables and equations of the design. Computational subprocesses were thereby highlighted. When such code was tested, one merely traced any discrepancy between the computed value and the symbolic value. Even before the code was checked out, manual run throughs detected design, code and consistency errors and allowed for full corrections. For the passive tracking subroutine integration, array field and register status maps were used in a similar manner. It is recommended that these maps be used in future applications on STARAN to contain the many opportunities for error and reduce checkout time.

References

[1] Feldman, J.D. and Reimann, O.A. "RADCAP: An Operational Parallel Processing Facility", Proceedings of the Sagamore Conference on Parallel Processing, August 1973.

[2] Goodyear Aerospace Corporation, STARAN Macro-Apple (MAPPLE) Programming Manual, GAC Document Number GER-15643A, September 1974.

[3] Goodyear Aerospace Corporation, STARAN User's Guide, GAC Document Number GER-15644A, September 1974.

[4] Lee, H.F., Katz, R., Fisher, T., Schenfele, F., Thomas, G.W., Associative Processor Application Study, Final Report, Volumes I and II. Contract No. F30602-75-C-0112, BCS Document Number BCS-40106-1 and -2, to be released.

[5] Prentice, B.W., "Implementation of the AWACS Passive Tracking Algorithms on a Goodyear STARAN", Proceedings of the Sagamore Conference on Parallel Processing, August 1974.

[6] Prentice, B.W., Katz, R., Komajda, R., Lee, H.F., Nelson, N., Associative Processor Application Study, Final Report. Contract No. F30602-74-C-0025, Report No. RADC-TR-74-326, September 1974.

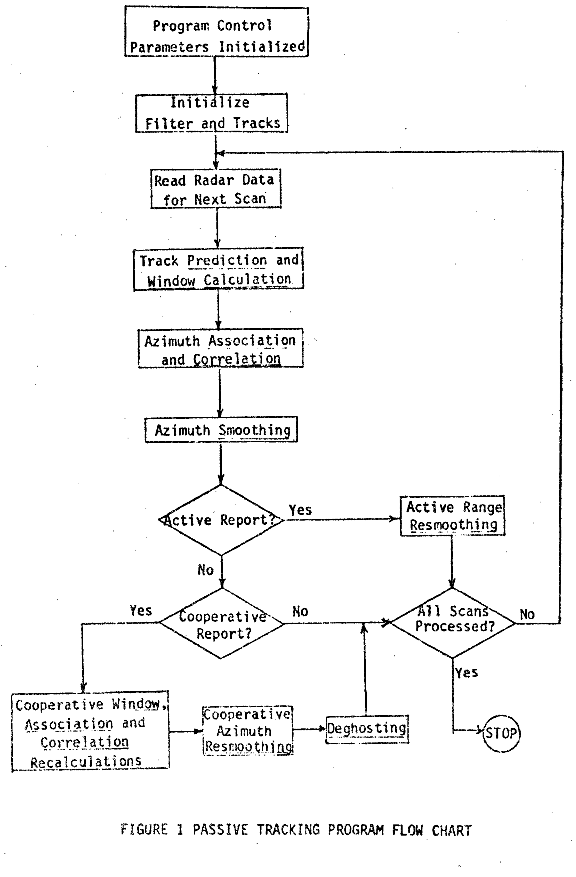

Program Control Parameters Initialized
Initialize Filter and Tracks
Read Radar Data for Next Scan
Track Prediction and Window Calculation
Azimuth Association and Correlation
Azimuth Smoothing
Active Report?
Yes
No
Active Range Resmoothing
Cooperative Report?
Yes
No
All Scans Processed?
No
Yes
STOP
Cooperative Window, Association and Correlation Recalculations
Cooperative Azimuth Resmoothing
Deghosting

FIGURE 1 PASSIVE TRACKING PROGRAM FLOW CHART

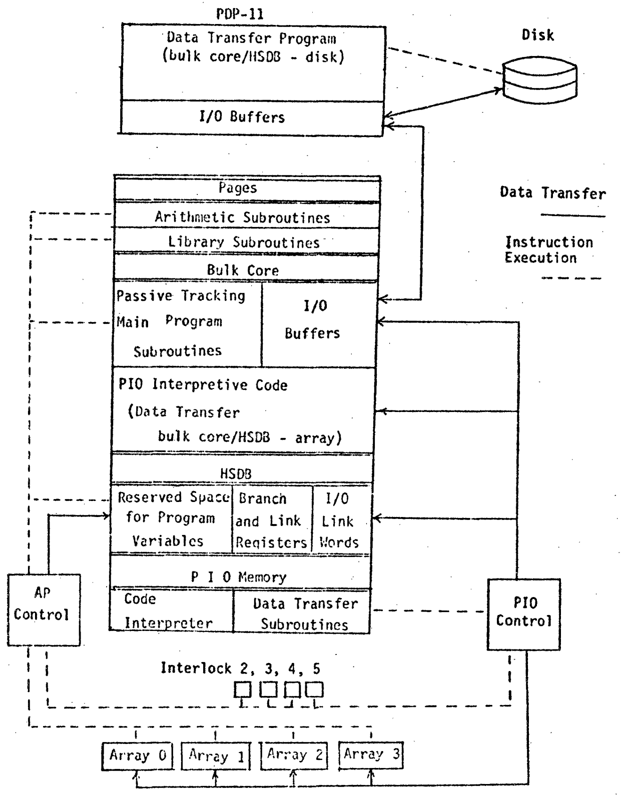


FIGURE 2 PARALLEL PROGRAM ALLOCATION & THEIR INTERRELATIONS

TABLE 1 FUNCTION OF INTERLOCKS

Interlock	Function
2	AP operation flag AP1 Reset 0 - AP running Set 1 - AP idling
3	AP array assignment flag AP2 Reset 0 - assignment of arrays complete Set 1 - assignment of arrays incomplete
4	PIO operation flag PIO1 Reset 0 - PIO running Set 1 - PIO idling
5	PIO array assignment flag PIO2 Reset 0 - Array Assignment complete; PIO operation in progress Set 1 - Array Assignment cannot proceed

TABLE 2 QUANTITATIVE STATISTICS FOR FLOATING POINT SUBROUTINES

FUNCTION	ACCURACY	TIMING	STORAGE
Cosine	Actual error $<2^{-14}$	Bulk Core: 0.0074768 sec.	258_{10}
Arctangent	Actual error $<2^{-16}$	Page Memory: 0.0027419 sec.	732_{10}
Absolute Value (Complement)	exact	Page Memory: 96.6 μ sec*	62_{10}
Field to Field Comparison (Less Than, Greater Than)	exact	Page Memory: 50.7 μ sec	57_{10}

* Add 30 μ sec. if the output field is mutually exclusive of the input field.

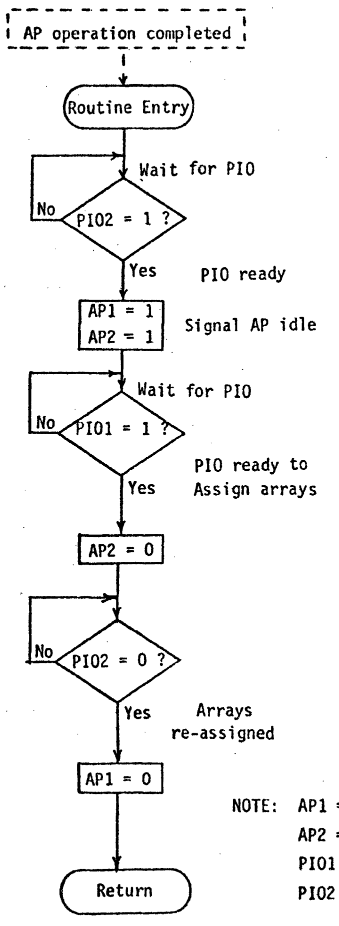


FIGURE 3a FLOW DIAGRAM OF AP SYNCHRONOUS SUBROUTINE

NOTE: AP1 = Interlock 2
AP2 = Interlock 3
PIO1 = Interlock 4
PIO2 = Interlock 5

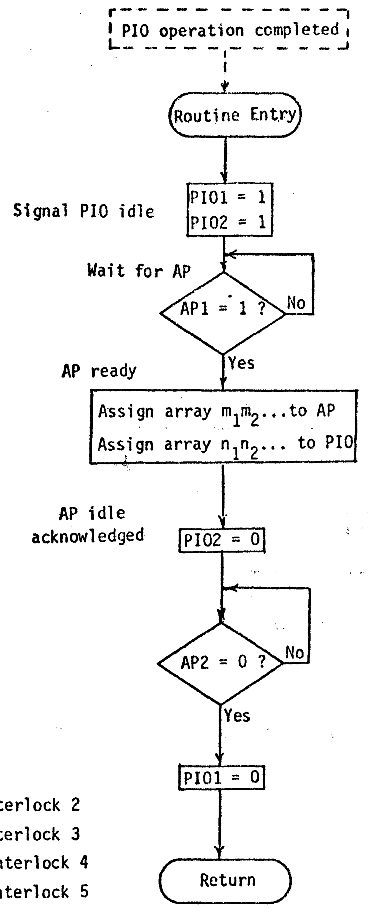


FIGURE 3b FLOW DIAGRAM OF PIO SYNCHRONOUS SUBROUTINE

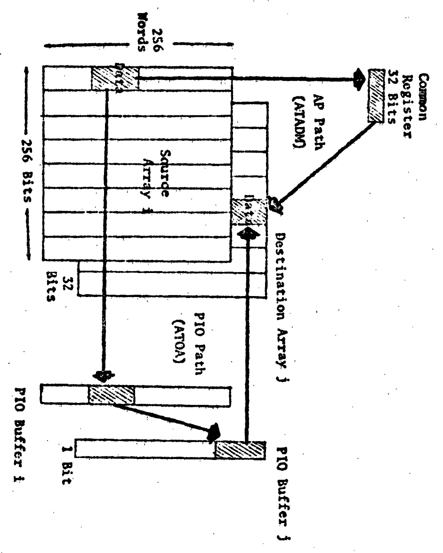


FIGURE 5 INTER-ARRAY DATA MOVEMENT PATHS FOR AP AND PIO

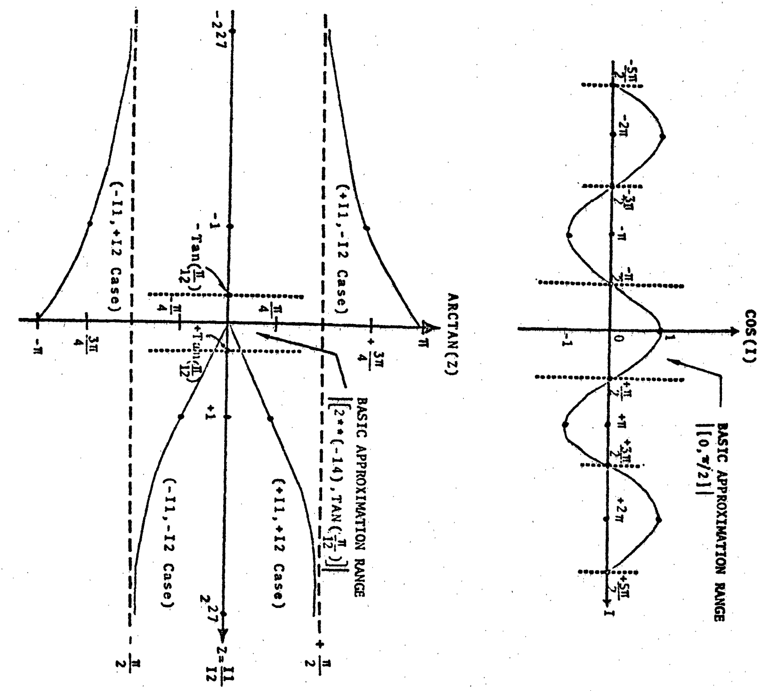


FIGURE 4 GRAPHS OF COSINE(I) AND ARCTANGENT(Z)

TABLE 3 AP AND PIO INTER-ARRAY DATA MOVEMENT TIMING

AP DATA MOVEMENT ROUTINE: MEASURED TIME

ATADM	NO. OF CALLS	TOTAL TIME (sec) OVER 2 SCANS	AVERAGE TIME (μ sec) PER CALL	AVERAGE NO. OF 32 BIT WORDS TRANSFERRED
Mode 1	59	0.0209263	354.683	58.7
Mode 0	114	0.0522385	458.232	76.0
Mode 2	93	0.0414003	445.165	74.0

AP AND PIO DATA MOVEMENT ROUTINES: COMPUTED TIME

NO. OF 32 BIT WORDS TO TRANSFER	AP: ATADM (μ sec)	PIO: ATOA (μ sec)	
		UNSHIFTED[(a)]	MAXIMUM SHIFTED
32	194.04	271.85	1209.05
64	386.68	276.25	940.25
128	771.96	288.85	671.45
256	1542.52	301.25	301.25

(a) Indicates identical location (no vertical shift) in source and destination arrays.

TABLE 4: SEQUENTIAL TEST CASE PERFORMANCE TIMES

Test Case	No. of Tracks	Measured CPU Problem State Time (seconds)	Summed Component Time (seconds)
1	64	14.74269	14.74530
2	52	11.58335	11.58556
3	40	8.62145	8.62368
4	28	5.85859	5.86001
5	16	3.31928	3.32033
6	32	11.62554	11.62744
7	24	8.53559	8.53722
8	16	5.62010	5.62152
12	64	18.45357	18.45616

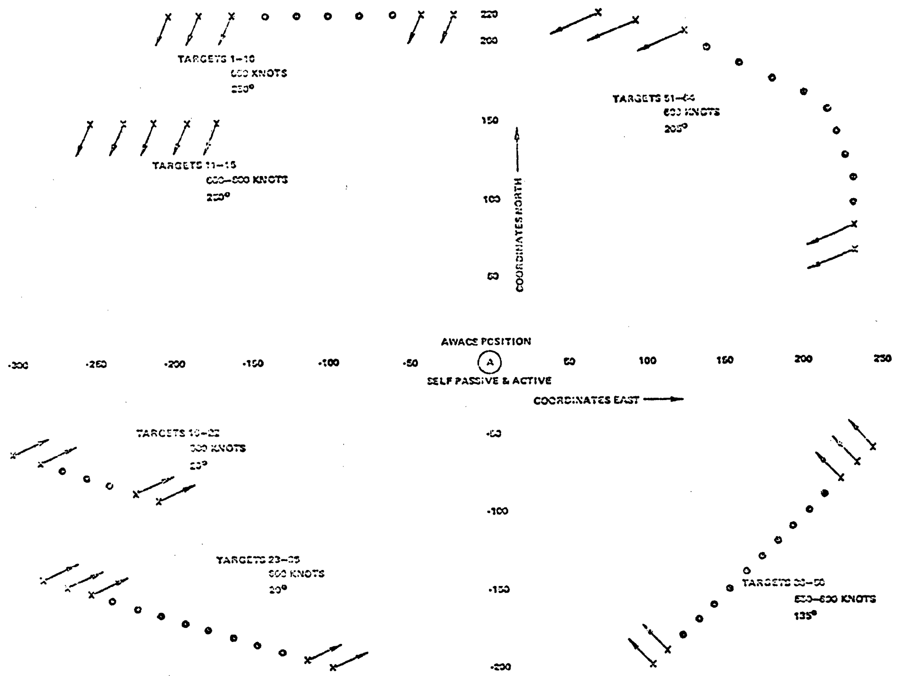


FIGURE 6 SCENARIO FOR SELF PASSIVE CASES 1-5 AND ACTIVE CASE 12

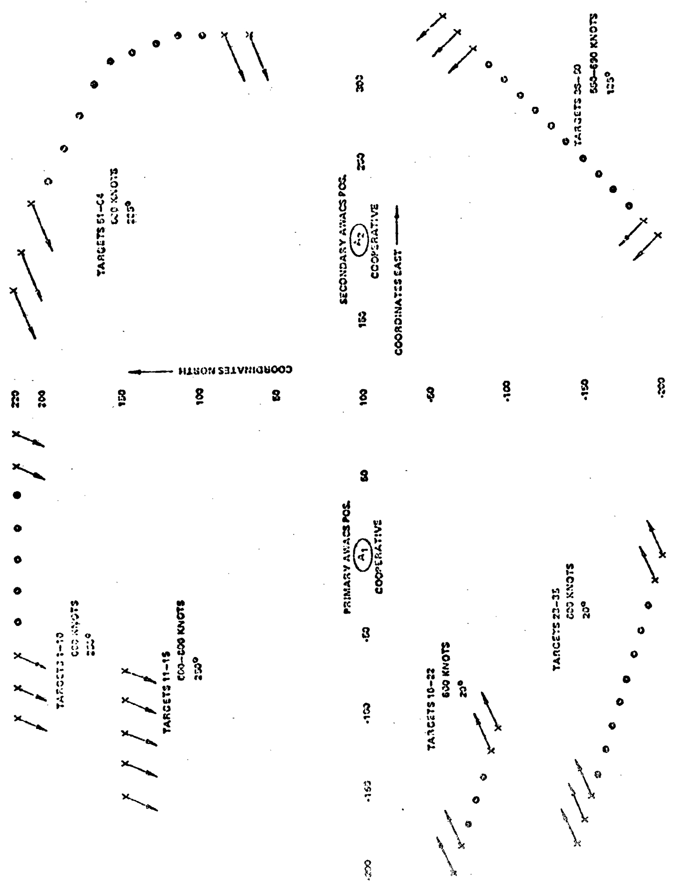


FIGURE 7 SCENARIO FOR CASES 6,7,8 - COOPERATIVE PASSIVE TRACKING

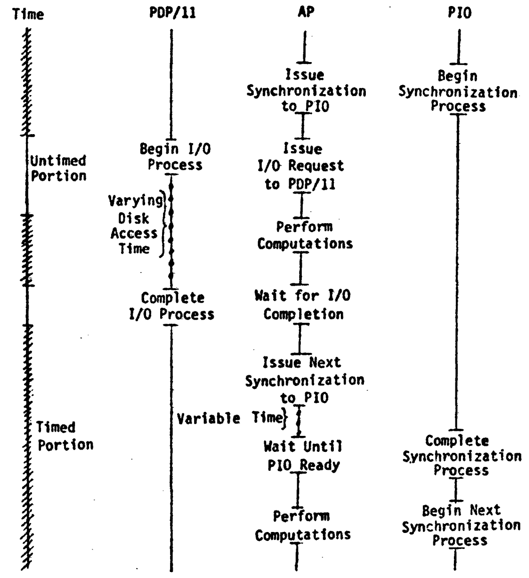
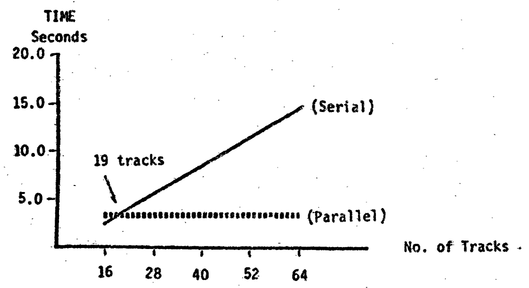


FIGURE 8 TYPICAL AP, PIO, PDP/11 EVENT STREAM SEGMENT DURING PASSIVE TRACKING EXECUTION

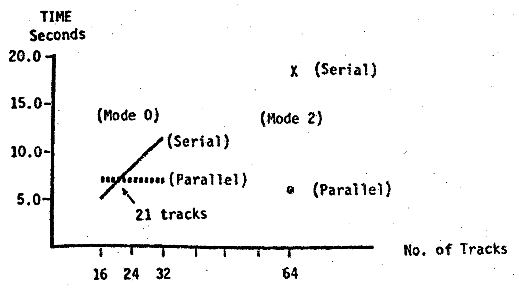
TABLE 5 PARALLEL/SERIAL VALIDATION AFTER 60 SCANS

				IRREGULARITIES (X,Y) (IN MILES)		
CASE NO.	TOTAL NO. OF TRACKS	MAXIMUM COMPONENT VARIATION (IN MILES)	TRACK NUMBER	DIFFERENCE OF PARALLEL TO SERIAL POSITION	DIFFERENCE OF PARALLEL TO TRUE POSITION	DIFFERENCE OF SERIAL TO TRUE POSITION
1	64	0.69	9	(3.47, 5.61)	(6.6, 10.35)	(3.13, 4.74)
2	52		5	(2.3, 2.12)	(12.31, 11.39)	(10.01, 9.27)
			6	(1.45, 1.55)	(3.33, 3.14)	(1.88, 1.59)
			7	(2.54, 3.27)	(10.69, 13.79)	(8.15, 10.52)
			8	(1.33, 2.20)	(2.66, 4.15)	(1.33, 1.95)
			9*	(42.7, 6.42)	(2.74, 0.22)	(45.44, 6.64)
			41	(1.23, 8.63)	(3.17, 24.04)	(1.94, 15.41)
			42	(0.21, 2.65)	(0.22, 17.45)	(0.01, 14.8)
			52*	(1.3, 0.24)	(5.9, 1.11)	(7.2, 1.35)
3	40		9*	(34.31, 10.98)	(2.49, 1.02)	(36.8, 12.0)
			25	(2.51, 6.8)	(4.17, 11.88)	(1.66, 5.08)
			26	(2.44, 4.65)	(4.66, 9.13)	(2.22, 4.48)
			27	(1.77, 1.71)	(6.71, 6.91)	(4.94, 5.2)
			28	(1.68, 1.31)	(6.53, 5.08)	(4.85, 3.77)
4	28	0.30	9	(19.87, 3.0)	(17.47, 2.81)	(2.40, 0.19)
5	16	0.22	9	(535.29, 9.82)	(535.99, 15.78)	(0.7, 5.96)
6	32	0.77	8*	(67.67, 8.71)	(1.68, 0.44)	(65.99, 8.27)
			9	(5.39, 2.72)	(6.73, 3.39)	(1.34, 0.67)
			15	(1.35, 0.37)	(1.55, 0.59)	(0.2, 0.22)
7	24	0.54	--	--	--	--
8	16	0.64	8	(1.8, 0.73)	(1.68, 0.9)	(0.12, 0.17)
12	64	0.0033	--	--	--	--

* Parallel position is significantly better than serial position.



Mode 1 (Self-passive tracking)



Mode 0 (Cooperative passive tracking) and Mode 2 (Active tracking)

FIGURE 9 PARALLEL AND SERIAL TIMING COMPARISON

TABLE 6 OVERALL TIMING - PARALLEL VERSION

		TIMING (Seconds) THROUGH 2 SCANS					TIMING (Seconds) THROUGH 60 SCANS			
CASE	NO. OF RUNS	AVERAGE TIME	LOWEST TIME	HIGHEST TIME	SUBROUTINE SUM TIME	NO. OF RUNS	AVERAGE TIME	LOWEST TIME	HIGHEST TIME	SUBROUTINE SUM TIME
1	7	0.0687	0.0566	0.0960	0.0629	6	3.7642	3.4786	4.1838	3.8422
2	8	0.0697	0.0569	0.0828		13	4.0062	3.5001	4.2418	
3	2	0.0662	0.0628	0.0695		2	4.0323	3.7590	4.3057	
4	2	0.0729	0.0698	0.0761		2	3.7594	3.5265	3.9922	
5	2	0.0730	0.0631	0.0830		2	3.7702	3.6788	3.8615	
(MODE 1)	(21)	0.0697				(25)	3.9116			
6	6	0.1259	0.0937	0.1589	0.1593	5	7.1616	6.5271	7.8350	7.4971
7	4	0.1251	0.1217	0.1286		4	7.0976	6.4113	7.8111	
8	2	0.1349	0.1283	0.1416		2	7.2532	6.5531	7.9534	
(MODE 0)	(12)	0.1271				(11)	7.1550			
(MODE 2)										
12	7	0.1075	0.0729	0.1533	0.1319	5	5.5120	5.0834	6.1637	5.4108

TABLE 7: AP WAITING TIME FOR PIO BY MODE

Mode	Waiting Time For 60 Scans (seconds)	Total Execution Time (seconds)	Percentage
Self passive	1.47424	3.9116	37.7
Cooperative	2.70486	7.1550	37.8
Active	2.30377	5.5120	41.8

Automatic Track Initiation Using the RADCAP STARAN

Edward C. Stanke, II, Capt, USAF
Rome Air Development Center
Griffiss AFB NY 13441

Summary

Automatic track initiation, as used in this paper, refers to the computer controlled initiation of new tracks within an aircraft active tracking environment. Aircraft active tracking consists of those steps necessary to keep track of the path of an airplane and predict the flight path for the next scan. This process is comprised of three basic steps. First, the reports which are received from the radar must be correlated with the appropriate tracks. Second, the stored present position of the track must be updated based on the report and the previous position. Third, a projected position of the track for the next scan must be predicted to prepare for the next correlation step. I will refer to these steps as association/correlation, smoothing, and prediction respectively. Notice that all three of these steps assume that the track has been previously established. That is, at some point in time, something performed those steps necessary to get the tracking procedure started on each track. In the type of active tracking previously implemented on the RADCAP STARAN (ITAS program [1], this initialization was done prior to the execution of the actual tracking program and no new tracks were added nor were any tracks deleted during the life of the tracking program. In an operational active tracking environment, the tagging of reports to be initialized as new tracks by some track initiation program is done by a human operator. Based on the demonstrated potential of the parallel tracking algorithm previously implemented on the STARAN, with the potential of tracking hundreds of tracks, the number of operators necessary for the initiation process becomes astronomical. This leads logically to the concept of automatic track initiation.

Automatic Track Initiation (ATI) consists of those steps necessary to recognize that a report is potentially an actual track when it does not correlate with any active track, and to perform the initialization process on that report to determine if it is in fact a valid track. This means that any report which does not correlate with an established track must be considered a potential track and must be saved at least until the next scan to test for correlation with received reports in that scan. In addition, ATI implies the deletion from the active track list of all those tracks which do not have reports correlated with them for some given period of time. The mechanism whereby this is done is a figure of merit which is assigned to each track. This figure, which I will call firmness, reflects a confidence level in the accuracy of the track. In the ATI implementation which I will describe, the range of the firmness is zero to seven. A firmness of zero is used as a space holder. That is, if a given track has a firmness of zero, that track is not active and the space it occupies is available for any potential track. The firmness values of one and two are used in the identification and initialization of new tracks. A firmness of three or greater means that the track is active. This eight-state system is similar to the ten-state firmness system described by Eddey and Meilander [2].

The implementation of the ATI concept using the previously implemented ITAS program requires the breaking up of the program into identifiable modules (basically the three steps identified above) and adding additional modules to take care of the track initiation function. The scheme to be used is fairly simple. All returned reports are to be checked for association with the established tracks. Then, all reports will be checked for association with all potential tracks which were started last scan. Both of these steps are sequential on the tracks and parallel on the reports. Next, all reports which were not associated with either a potential or established track are identified as potential tracks for the next scan. All potential tracks which did not have a report associate with them are dropped. Following this, all potential tracks which have a report associated with them are initialized in parallel and upgraded to established tracks for the next scan. Previously established tracks are then tested for correlation and smoothed via a Kalman filter. Finally, all tracks, established and potential, go through the prediction routine in preparation for the next scan. Note that this sequence is the same as the parallel ITAS flow with the ATI loop added. Therefore, the effect on execution time of the ATI step can be fairly well characterized in terms of the scenario characteristics.

Using the results from the ITAS implementation [3], the parallel execution time per scan with ATI can be given by:

$$\text{Time per scan} = 64.4 + 0.4N + F/100 + \text{ATI msec}$$

Where: N = number of established tracks
E = number of false alarms
ATI = time due to automatic track initiation

It is easy to perceive that the ATI term in the above equation consists of two terms. One of

these is a constant which represents the time required for updating of the potential tracks which do have a report associated with them, plus the time required for initialization of all unassociated reports as potential tracks. Since this updating and initialization are trivial compared to the correlation and smoothing which account for the constant 64.4 in the above equation, this term is probably negligible. The second term making up ATI is the one accounting for the potential track association loop. Since this loop is very similar to the loop for established tracks, the ATI term will in all probability look something like:

$$ATI = 0.4NP$$

where NP is the number of reports which didn't associate on the last scan. This is, of course, highly dependent on the false alarm rate in the given environment.

From the above figures, it is apparent that for a 1000-target environment, with an equal number of false alarms per scan, the execution time for the tracking program with ATI is somewhat less than twice that for tracking alone. Based on projected figures for the ITAS program, the execution time per scan for tracking in this environment was 0.5 seconds. Thus, we could do the tracking with automatic track initiation in less than one second which is ten percent of the time available per scan. It then seems reasonable to attempt automatic track initiation on the STARAN to relieve the potentially large manual intervention necessary for the track initiation process in an operational environment.

References

[1] M.W. Summers and D.F. Trad, "The Evolution of a Parallel Active Tracking Program", Lecture Notes in Computer Science, Vol. 24, Springer Verlag Series Lecture Notes in Computer Science, pp 238-249.

[2] E.E. Eddey and W.C. Meilander, "Application of an Associative Processor to Aircraft Tracking", Lecture Notes in Computer Science, Vol. 24, Springer Verlag Series Lecture Notes in Computer Science, pp 417-428.

[3] M.W. Summers, "An Associative Processor Application Study", USAF, Rome Air Development Center, RADC-TR-75-318, January 1976, 43 pp.

CONCEPT FOR A COMPUTER ARCHITECTURE RESEARCH FACILITY

Alan R. Klayton, Capt, USAF
Rome Air Development Center
Griffiss AFB, Rome NY 13441

Summary

Problems such as the high cost of software, system availability and reliability, and the requirement for increased processing power continued to be of major concern. In some of these areas only software based solutions have received much attention; e.g., structured programming. Now, however, rapid progress in LSI technology, the availability of microprocessors, and the application of microprogramming techniques offer new opportunities for seeking hardware solutions to key data processing problems. Computer hardware concepts previously discarded mainly for economic reasons are now feasible. In fact, architectures of the future will most certainly be built from collections of microprocessors integrated in a manner suitable for the intended application. The increasing interest in concurrent processing systems is already evident with economics a positive driving force as microprocessors and memory prices continue to fall.

Unfortunately, rapid advances in LSI hardware technology have surfaced new problems. We lack techniques for efficiently designing and developing new architectures utilizing the new LSI building blocks. Methods for devising and optimizing multi-microprocessor architectures need to be developed. More specifically, there is a strong requirement for a computer architecture research facility where nonstandard architectures can be designed, evaluated and tuned to an intended application. To be practical, the facility must support the efficient emulation, application programming, and performance monitoring of a wide range of computer architectures.

In response to these needs, the Rome Air Development Center is assembling an experimental computer architecture research facility which will be a testbed for: 1) the identification and development of necessary facility hardware and software support tools and 2) exploring the application of multi-microprocessor architectures and system tuning techniques to the problem areas mentioned above.

Although the design of the facility is the subject of a number of studies, Figure 1 depicts the major hardware components available for the testbed and outlines a candidate interconnection scheme.

The QM-1 is an extremely flexible sequential computer featuring two levels of microprogrammability with both levels fully accessible to the user. The QM-1 itself offers a strong emulation capability for conventional architectures and is the main link to the other system components.

The STARAN Associative Processor is a representative of single instruction multiple data stream processing systems. This nonconventional architecture has been shown to provide great processing power for applications possessing a high degree of parallelism. The multiple dimensional access memory in STARAN is a unique memory organization capable of 256 different read/write modes. This flexibility allows optimized high bandwidth interfaces to exist between the array memory and the other component parts of the facility, such as a 256-bit slice or word slice to and from mass memory, 64 simultaneous four-bit words to and from the microprocessor array, eight simultaneous 32-bit words to and from the control system, etc.

Each of the microprocessors in the microprocessor array is an arithmetic and logic unit on a single chip. In today's technology, they are 2, 4, 8, 12, and 16 bits wide. Under program control, the microprocessor chips will be made to act as independent processing units or grouped together into functional units. Depending on the application, they may function as stages of a pipeline machine, peripheral controllers, more powerful processing elements, or elements in a distributed multiprocessor system. Since each of the microprocessor units can execute from its own program memory, they can perform autonomously or be made to operate on a cycle-by-cycle basis on instructions and data issued under control of the control computer.

The data manipulator routes data and instructions between the various elements of the system and perhaps most importantly, between microprocessors. It can be thought of as a software controlled switch which in effect creates a generalized high bandwidth interconnection between elements of the testbed.

The mass memory is a backup store for the STARAN multiple dimensional array and the microprocessors. It will hold data and instructions and be able to supply them quickly over the high bandwidth (1024 bits wide) channel connecting the STARAN arrays to the mass memory.

The Performance Monitoring System (PMS) is an essential element of the facility. The PMS is required to conduct quantitative analysis of alternative computer architectures and to perform hardware/software/firmware tradeoffs.

The PMS will employ both hardware and software measurement techniques.

The STARAN associative processor is operational at RADC. A mass memory prototype effort is currently in progress and the Data Manipulator is now under construction. The QM-1 is an off-the-shelf item. The microprocessor array and all system interfaces must be designed. The research facility will be interfaced to the MULTICS time-sharing system through the QM-1 in order to take advantage of the many services available on the MULTICS system, and to facilitate a multi-user mode of operation for the facility.

The success of the computer architecture research facility concept hinges on the development of those software tools required for achieving system practicality. The major tools presently identified, called the USER (Universal Software Emulation Resources) Package, is depicted in Figure 2.

The Emulation Design Language (EDL) is a high order language (HOL) used for describing the target architecture which is to be emulated. The EDL must provide appropriate constructs for describing concurrent processing systems.

The EDL translator will be a cross compiler running under the MULTICS system. The translator will output microcode, representing the target machine emulation, as well as a set of tables, etc., required by the automated code generator section of the application language (HOL) compiler.

Full realization of the RADC Computer Architecture Research Facility as described above requires hardware and software state-of-the-art advances. Initial RADC efforts will focus on the development of the USER Package depicted in Figure 2, but will be restricted to supporting the QM-1 emulation computer. As experience is gained, and as the results of various studies become available, the facility hardware and software will be expanded towards the system of Figure 1.

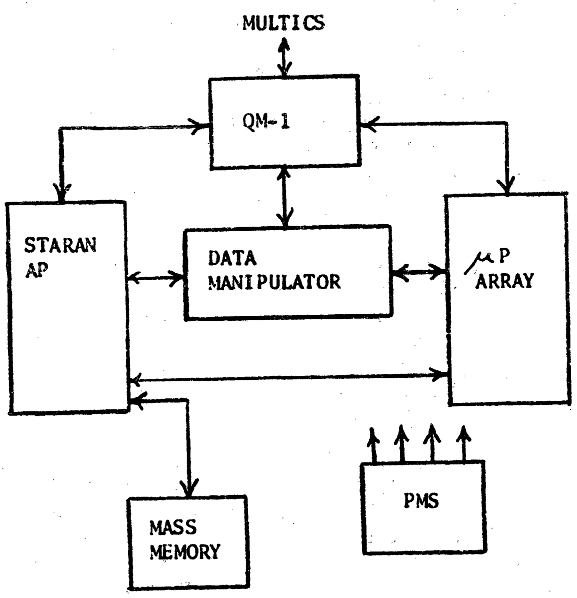


Fig 1 SYSTEM ARCHITECTURE RESEARCH FACILITY

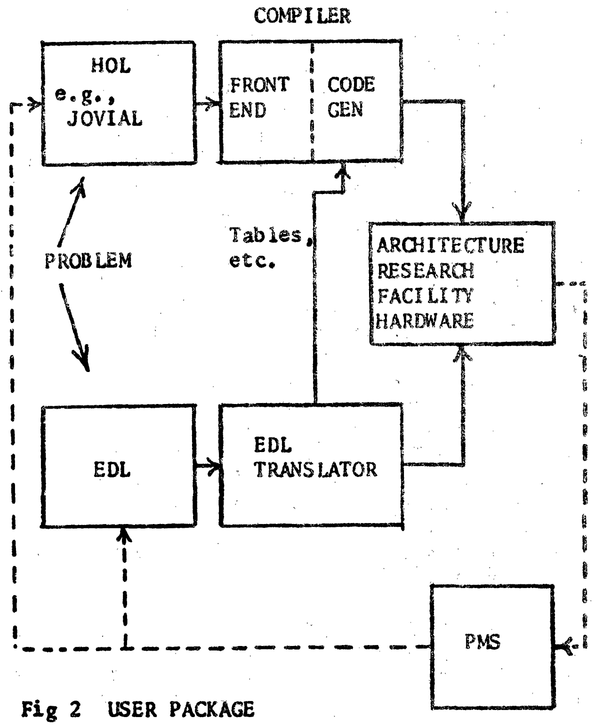


Fig 2 USER PACKAGE

A CONTENT-ADDRESSED MEMORY DESIGNED FOR DATA BASE APPLICATIONS[a]

George A. Anderson
Honeywell Systems & Research Center
Minneapolis, MN 55413

Richard Y. Kain
University of Minnesota
Minneapolis, MN 55455

Abstract—This paper describes an extended-capability CAM which operates on formatted data bases up to around 10^9 bits in size. The system, called ECAM, consists of a control unit and a serial associative store with logic blocks in the 100 gate complexity range provided at each word. The organization is relatively independent of storage technology; the current design assumes CCD memory organized in 4K bit words, shifted at a 1 microsec. rate. The control unit is microprogrammed to interpret a block-structured query language which operates on logical data structures, such as Codd's relations. The mapping of logical structures to operations on physical storage is performed by the hardware. A full repertoire of associative search and arithmetic operators is provided. In its target application, the ECAM will require less than 1 hour/day to perform a task that was estimated to require between 40 and 700 hours/day on a large commercial mainframe.

Introduction

Until recently, the extremely high cost of implementing associative memories and processors has restricted implementations to very small sizes. The serial STARAN and OMEN processors have capacities up to the 10^6 bit range [1], while the biggest fully parallel device is only 16K bits in size [2]. This paper describes a design, based on recent improvements in storage technologies, which allows implementation of memories up to the 10^9 bit range. The system, called an Extended Content-Addressed Memory (ECAM), is being developed by Honeywell, Inc. under the sponsorship of Rome Air Development Center, USAF, and is designed specifically for high-performance database applications. We first describe the motivation for such a machine, then the hardware and software approaches being used.

Background

The requirement for a device such as the ECAM stems from the inherent performance limitations in conventional database approaches. Conventional database systems are implemented on serial processors with limited amounts of fast memory. This has resulted in performance which deteriorates drastically as the database size increases. Also, conventional memories are location addressed, a fact which complicates the processing problem with issues not inherent in either the data or the system functional requirements. The major effect of location-addressing has been increased storage overhead for index tables. In large databases, management of these tables is a problem in its own right. Fast insertion and deletion of records requires that a minimum of tables be involved; fast retrieval of records requires that a large number of different attributes be indexed in the directories.

In contrast to conventional techniques, content-addressed memories like the ECAM have the capability of retrieving information directly, based on attributes of the data itself. This is accomplished by including sufficient processing capability in the data storage medium to perform searching operations. The use of Content-Addressed Memories (CAMs) to overcome the constraints of conventional database systems has been suggested by many (most recently by DiFiore [3]), but until recently the cost of CAM systems has been prohibitively high. This high cost was due both to the cost of logic required for content addressability and to the high cost of the storage itself. To date, 10^6 bits has been the upper limit on implemented CAMs, while our Air Force requirements called for capacity to approximately 10^9 bits. Advances in LSI technology, however, have reduced the cost of storage to the point where historical limits no longer apply. Accordingly, the ECAM work was undertaken with the objective of designing a buildable 10^9 bit associative memory by taking advantage of these developments.

Hardware Structure

The ECAM is a special-purpose machine designed to be attached to one or more host computers and to be used as an access processor for the database it contains. The major functional units of the ECAM are shown in Figure 1. An artist's sketch of the proposed packaging is shown in Figure 2.

The machine is divided into two portions; the CAM array, and the control unit. The control unit is designed around a bus-organized minicomputer and includes the mini (called the master), a custom-designed controller for the array (called the slave), and one (or more) interfaces to the host(s). The array consists of a multiplicity of 4K bit serial storage words, with combinational logic at each word to effect the associative functions. The upper limit on array size is approximately 10^9 bits.

Control Unit

The main unit within the control unit is the master minicomputer. Its memory bus provides the

[a] This is a preprint of a paper to appear in the Proceedings of the 1976 International Conference on Parallel Processing, held August 24-27, 1976.

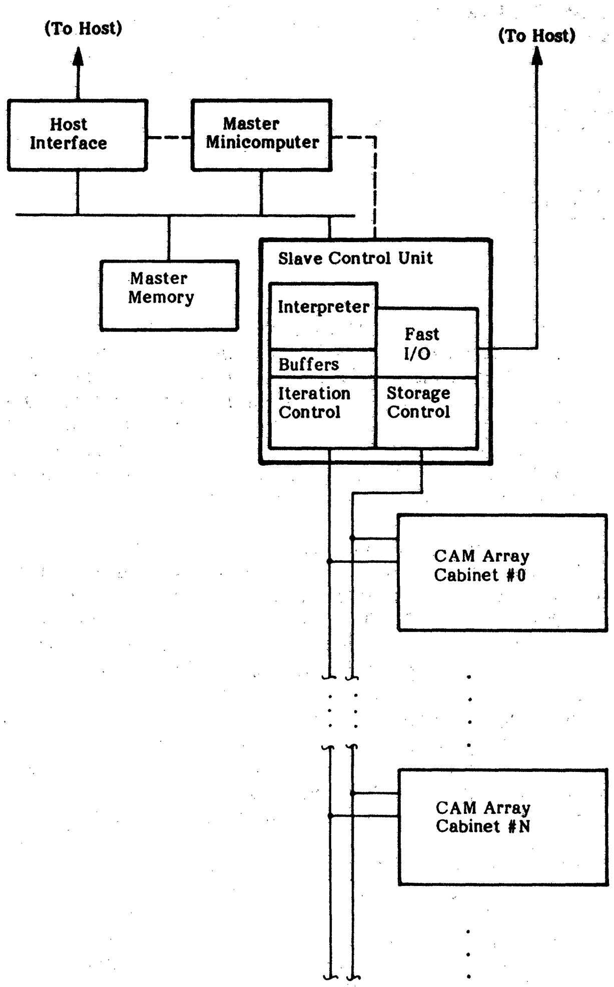


Figure 1. ECAM Structure

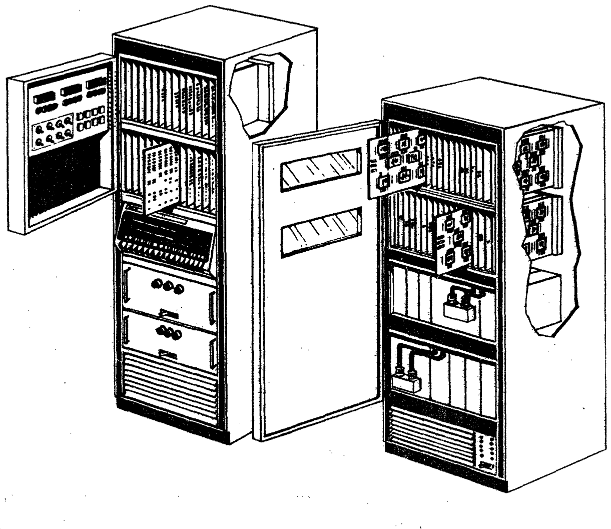


Figure 2. ECAM Packaging

basic structure of the control unit. Furthermore, the availability of standard software facilitates writing application code to mediate between the host and the slave controller. Because of its ubiquity in Air Force applications, we are recommending that the PDP-11/45 be chosen as master control processor, but the control unit design is such that almost any bus-organized minicomputer could be used.

The ECAM-host interface is designed to connect to the host as a standard high-speed peripheral (such as a disc). It is controlled via the master's programmed I/O facility and transfers blocks of information between the host and the master's memory in a transparent fashion.

The two major subunits of the slave are the interpreter and the iteration control. The interpreter is a high-speed microprogrammed unit which is designed specifically for interpretively executing a block-structured query language used to specify ECAM operation sequences. Query language sequences are passed from the master to the slave via buffers in the master's memory. The control store of the interpreter is writable, allowing easy changes to the query language. The output of the interpreter is a stream of array primitives which are passed to the iteration control unit via dedicated buffers within the slave. Iteration control is a hardwired subunit which generates control signal sequences to effect the array operations.

In order to keep the bulk of the design independent of the storage technology, the storage control functions of addressing, shifting, refresh, etc. have been isolated to a single subunit of the slave. A high-bandwidth I/O capability is also provided, under control of a distinct subunit of the slave.

The complexity of the slave controller is estimated at 500-800 small and medium-scale integrated circuit packages. It is designed for implementation in TTL circuit technology with 10 MHz clocks.

Array

The array consists of a large number (up to 250,000) of associative words, as shown in Figure 3. Each word consists of 4096 bits of CCD storage,

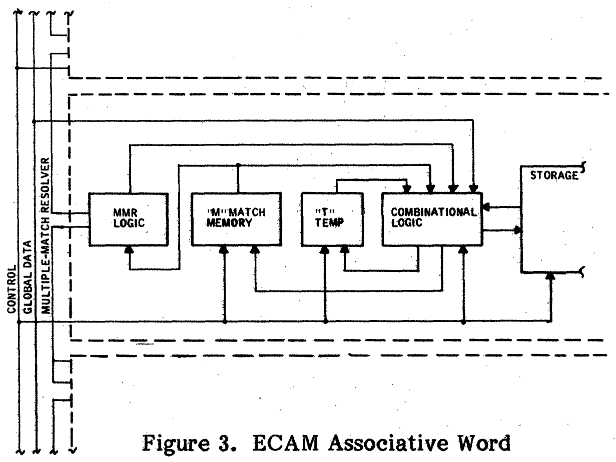


Figure 3. ECAM Associative Word

randomly addressable to 256 bit registers, and a block called the "word logic" which supports the content addressing and associative functions of the array. The two major elements of the word logic are the match memory and the arithmetic-logic block. Word logic operations such as searches, arithmetic, etc. are performed by selecting one of 16 match bits from the memory and repeatedly executing the same sequence of combinational operations on each bit of a field within the storage word. For most operations, the inputs to the combinational logic are the selected match bit, a "global" data signal from the control unit, and the "local" data bit from the storage part. A summary of word logic functions is shown in Table 1.

Table 1. Word Logic Function Summary

Processing

- Add/Subtract
- Reverse Subtract
- Arithmetic Compare
- Minimum/Maximum

Input/Output

- Input
- Output
- Output and Tag Duplicates

State Manipulation

- (14 Logical Functions between Match, T, and Other Word Logic State Variables)

I/O and MMR Control

- (4 Functions for I/O, MMR, and Match Counting)

The ECAM packaging baseline assumes that the storage is contained on LSICs of 10 words by 4096 bits. This is within the capability of present CCD technology of many vendors. In the baseline, we have also assumed a shift rate of 1 microsec/bit. This is somewhat slower than current technology capabilities, but was chosen to simplify signal distribution. Each storage chip is paired with a word logic chip containing ten word logic blocks together with first-level support logic for the multiple-match resolver, match counting, and I/O facilities. At the next level of packaging, eight storage/word logic pairs are mounted on hybrid substrates. These substrates are then placed on conventional circuit cards. An artist's sketch of the packaging scheme is shown in Figure 4.

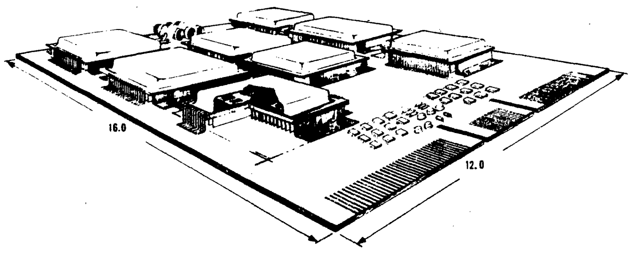


Figure 4. ECAM Array Packaging Scheme

Although the baseline ECAM storage technology is CCD, the design has deliberately been kept as technology independent as possible. The basic constraint on the storage medium is a requirement for stop/start and read-modify-write capability on a per-bit basis. Beyond this, the speed, cost, power consumption, and mechanical attributes of the ECAM may be varied by changes in the storage technology. In particular, use of magnetic bubble storage would allow ECAM sizes to increase to perhaps 10^{10}-10^{11} bits with a corresponding reduction in speed.

In addition to the word logic shown in Figure 3, the ECAM is provided with a high speed I/O path which allows 10 words to be logically selected onto I/O lines and participate simultaneously during a single input or output operation. The switch which implements this fast I/O mode is included at the word logic chip level. The effective transfer bandwidth of the ECAM is raised from 10^6 bits/sec to 10^7 bits/sec by use of the fast I/O mode. This faster bandwidth capability is required for database checkpoint/restart operations.

Application and System Software

The ECAM is primarily intended to operate on data stored as tables consisting of a number of fixed-size records, each subdivided into fields of varying length. The design is consistent with the relational view of data [4], where the tables are the relations, the records are the n-tuples, and the fields contain domain values. An example of this basic structure is shown in Figure 5. Both binary integers and characters are used as field values; within the ECAM, they are treated uniformly as bit strings.

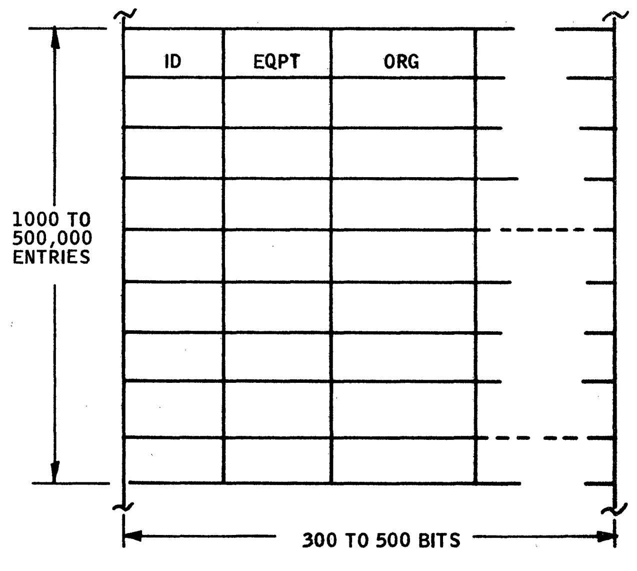


Figure 5. Stored Data Structure

The division of functions between host and ECAM is made as follows; terminal handlers and user job interfaces within the host support the generation of query sequences. Once such a sequence has been prepared, it is transferred, together with identification tags, to the ECAM via the host's standard I/O subsystem hardware and software. The query sequence references the logical structure of the data stored in the ECAM and may, in addition, refer to intermediate search results left by the user after previous queries.

The master control processor is responsible for management of pending sequences and for transmitting the results of queries back to the host. This includes allocation of master memory buffers for incoming sequences, scanning of sequences to create code for the interpreter, and allocation of master memory buffers for results being returned by the slave. In addition, users requiring temporary storage of results during the period between two queries may request temporary storage areas within the ECAM. The master control processor manages these areas. The master also schedules and dispatches code blocks to the interpreter. The slave is "multiprogrammed" —code blocks may include "WAIT" type operations which relinquish control, but no preemption is allowed.

The mapping of the logical table structures onto the physical storage is shown in Figure 6. A small number of physical word formats are defined, each having a different combination of directory entries. A word may contain one or more entries from one or more of the directories. The particular packing strategy chosen will result from a tradeoff between speed and storage efficiency for a given set of directory widths and lengths. The choice of a packing strategy is a database administration function; changes are expected to occur infrequently.

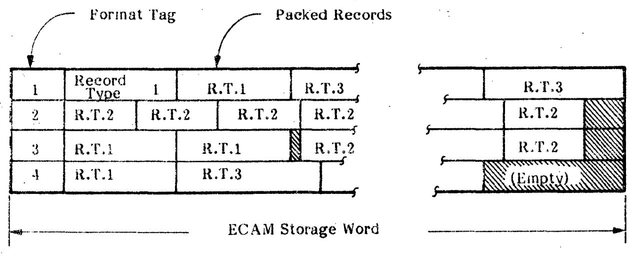


Figure 6. Logical → Physical Mapping

The ECAM word formats are defined by a set of descriptor tables stored in the master's memory. These consist of: (1) a Record Type table listing attributes of the various records (n-tuples) and pointing to (2) Record Instance lists describing alternative physical placements of each record type. Finally, a Field Descriptor table provides information on the placement of fields within records.

The function of the interpreter is to execute the code sequences received from the master, including transforming references to the logical data structures into references to the physical storage scheme by use of the descriptor tables. This involves creation of loops to sequence through multiple instances of records and modification of programmer-specified loops to optimize shifting of the array.

An example of the slave's language is shown in Figure 7; the program marks that record containing the maximum value in FIELD3 among those records of type TYPE1 in which either FIELD1 contains VALUE1 or FIELD2 contains VALUE2. This language has the following characteristics:

```
FOR ALL RECORDS (TYPE1) DO
    FIND(VALUE1,FIELD1)
    PUSH1
    FIND(VALUE2,FIELD2)
    OR
    MAXIMUM(FIELD3)
ENDFOR
```

Figure 7. Example Intermediate Language Sequence

1. It is strictly block-structured and can be interpreted using a single stack.

2. The programmer sees the data in its relational form; there may be many instances of "TYPE1" records over which the operations within the "FOR" block must be repeated; these instances may occur in various word formats and various positions, as specified by the descriptor tables.

3. The programmer sees the match memory (Figure 3) as a stack (between operations, these stacks must be saved with the record instances, since they are part of the process state).

The syntax and semantics of the intermediate language are completely defined by the microcode stored in the slave. No part of the language interface has been hardwired. Rather, the hardware has deliberately been kept general, so that experience with the operational system can be used to make improvements in the master/slave language interface.

The software is structured and its functions divided among the system components in a hierarchical manner: the host need not know any details of the database structure or the query language implementation; the master need not know details of the iterations through bit positions to scan fields; the iteration controller handles these lowest-level details in a predetermined way. By this, we feel we have succeeded in moving complexities to the lowest possible level, while simultaneously designing to allow change in all critical areas.

Performance

Analytical estimates of ECAM performance indicate that it will operate roughly 200 times as fast as a conventional database system on a large commercial mainframe. We estimate that the ECAM will be significantly less than 10% loaded in an environment where the conventional machine would be overloaded from 100 to 2000%.

References

[1] K. J. Thurber and L. D. Wald, "Associative and Parallel Processors," ACM Computing Surveys, (December 1975), p. 197.

[2] L. D. Wald and G. A. Anderson, Associative Memory for Multiprocessor Control, Final Report No. NAS12-2087 (September 1971). Also IEEE-CS Repository No. R74-172.

[3] C. R. DiFiore and P. B. Berra, "A Quantitative Analysis of the Utilization of Associative Memories in Database Applications," IEEE Transactions on Computers, (February 1974), p. 121.

[4] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," CACM, (June 1970), p. 377.

TIME AND PARALLEL PROCESSOR BOUNDS FOR LINEAR RECURRENCE SYSTEMS WITH CONSTANT COEFFICIENTS *

S. C. Chen
Advanced Technology
Federal and Special Systems Group
Burroughs Corporation
Paoli, Pennsylvania 19301

Abstract. Parallel and direct computational algorithms are developed to evaluate linear recurrence systems with constant coefficients. We show that $O(\log_2 m \log_2 n)$ time steps and $O(mn)$ processors are sufficient to solve such a system. We also show that general recurrences, i.e., with m=n, can be computed within $O(\log_2^2 n)$ time steps with at most $O(\frac{n^2}{4})$ processors.

All algorithms are aimed at easy data routings and simple machine control structures. Thus, they can be easily implemented through software such as parallel compiler algorithms, numeric subroutines, or hardware control programs for future parallel or pipeline processors.

1. Introduction

Linear recurrences with constant coefficients arise frequently in general numerical computations. Several analytic methods for the solution of these equations are available, such as by solving for the roots of its characteristic polynomial or by the use of generating functions.

We are interested in a parallel and direct computational algorithm which can be used to evaluate them at a high speed. Such systems may be represented as $\bar{x} = \bar{c} + A\bar{x}$ where $\bar{c}$ is a constant column vector, A is an nxn strictly lower triangular matrix with bandwidth of m, $m < n$, and all elements are identical along each sub-diagonal. We show that $O(\log_2 m \log_2 n)$ time steps and $O(mn)$ processors are sufficient to solve such a system. We also show that general recurrences, i.e., with $m = n$, can be computed within $O(\log_2^2 n)$ time steps with at most $O(\frac{n^2}{4})$ processors. While such systems remain in the same order of speed as in the algorithms for arbitrary coefficients discussed in [1], the bounds on processors are sharpened by a factor of m and n for the mth order and general constant coefficient system, respectively. To achieve these results, all processors need only perform one type of operation at each time step (SIMD operation).

We further show that our method can be modified to evaluate the remote terms in a homogeneous linear recurrence sequence with at most $O(m^2)$ processors in contrast to the $O(m^3)$ processors required by the Miller-Brown algorithm. Also, the parallel evaluation of nth degree polynomials can be completed within $2\log_2(n+1)$ time steps with at most $\lceil\frac{n+1}{2}\rceil + 1$ processors which compares favorably with recent results [2], [3], for practical applications.

The parallel evaluation of recurrence relations has been studied by a number of people [4], [5], [6], [7], [8], in addition to the most recent results [1] as stated above. For constant coefficient systems, however, the algorithms presented here will provide more efficient computations than all previous results.

Although we do not discuss any details of machine organization in this paper, it is in order to sketch a machine here. We assume that:

1. Instructions are always available for execution as required and are never held up by a control unit.
2. Each processor may perform any of the four arithmetic operations in one unit step.
3. There are no memory or data alignment time penalties. Most of these assumptions can be approached in a properly designed system as discussed in [9].

The following definitions will hold throughout the paper. Let T_p be the time, measured in unit steps, required to perform some calculation using p independent (parallel or pipeline) processors. We define the speedup of a p-processor machine over a uniprocessor as $S_p = \frac{T_1}{T_p}$, and we define its efficiency as $E_p = \frac{T_1}{pT_p} \leq 1$, which may be regarded as the quotient of S_p and the maximum possible p-processor speedup p. For notational simplicity, we will assume all logarithmic functions take their ceiling values.

2. General Linear Recurrences with Constant Coefficients

In this section we discuss bounds on the time and processors for the direct evaluation of the following class of general linear recurrences

*This research is supported by NSF Grant GJ-36936 while the author was with the Dept. of Computer Science, Univ. of Illinois, Urbana, Illinois.

$$R\langle n\rangle: \quad x_i = 0 \qquad \text{for } i \le 0,$$
$$= c_i + \sum_{j=1}^{i-1} \alpha_j x_{i-j} \quad \text{for } 1 \le i \le n .$$

In matrix notation, we write this as $\bar{x} = \bar{c} + A\bar{x}$ where $A = [a_{ij}]_{n\times n}$ is a strictly lower triangular matrix. For example,

$$R\langle 4\rangle: \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ \alpha_1 & 0 & 0 & 0 \\ \alpha_2 & \alpha_1 & 0 & 0 \\ \alpha_3 & \alpha_2 & \alpha_1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

For simplicity, we will assume n is a power of 2. The main result is Theorem 1. We also give Algorithm 1 which may be used as a basic algorithm for the parallel evaluation of this class of problems.

First, let us state one important lemma which will be fundamental in obtaining the main results.

Lemma 1 Given any

$$R\langle n\rangle: \quad \bar{x} = \bar{c} + A\bar{x} ,$$

there is an associated Y matrix such that

$$\bar{x} = \bar{c} + Y\bar{c} ,$$

in which $Y = [y_{ij}]_{n\times n}$ is a strictly lower triangular matrix and

$$\bar{y}_j = \bar{a}_j + A_j\bar{y}_j \qquad \text{for } 1 \le j \le n \qquad (1)$$

where $\bar{y}_j = (y_{j+1,j}, y_{j+2,j}, \ldots, y_{n,j})^t$,

$\bar{a}_j = (a_{j+1,j}\ a_{j+2,j}, \ldots, a_{n,j})^t$,

$$A_j = \begin{bmatrix} 0 & & & & \\ a_{j+2,j+1} & 0 & & & \\ \cdot & \cdot & \cdot & & \\ \cdot & & \cdot & \cdot & \\ \cdot & & & \cdot & \cdot \\ a_{n,j+1} & \cdot\ \cdot\ \cdot & & a_{n,n-1} & 0 \end{bmatrix}$$

Proof: This is a corollary of Lemma 1 in [1], hence the proof will be omitted here.

Since all elements along each sub-diagonal of A are the same, it follows directly from equation (1) that

Lemma 2 Given any

$$R\langle n\rangle: \quad \bar{x} = \bar{c} + A\bar{x} ,$$

its associated matrix Y has identical elements along each sub-diagonal. For example, let

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \quad \text{then } Y = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 8 & 3 & 1 & 0 \end{bmatrix}$$

It is obvious that the j th column of Y can be obtained by simply shifting the first column (j-1) places downward. We will denote this operation as $\bar{y}_j = \bar{y}_1[j-1]$ where $\bar{y}_j$ is the j th column of matrix Y. These notations will be used throughout this chapter.

Now, we present the proposed parallel algorithm and the proof of its effectiveness will immediately follow.

Algorithm 1

a) Let B be a lower triangular matrix of order (n×n) in which

$\bar{b}_1 = (c_1, c_2, \ldots, c_n)^t$,

$\bar{b}_2 = (0, \alpha_1, \alpha_2, \ldots, \alpha_{n-1})^t$,

and $\bar{b}_j = \bar{b}_2[j-2]$, $3 \le j \le n$.

b) Let C be an alias for B, i.e., B and C represent the same memory locations.

c) Repeat this step for i = 1, 2, ..., $\log_2 n$;

```
i:   Set $k = 2^i$;
ii:  Partition B and C as shown in
     Figure 1;
iii: Compute $S_j = S_j + T_j * Q_j$
     for $1 \le j \le \min(2, \frac{n}{k})$
     simultaneously;
iv:  Set $\bar{y}_j^{(i)} = \bar{y}_1^{(2)}[j-1]$
     for $1 \le j \le k$, $2 \le i \le \frac{n}{k}$ ;
v:   Compute $D_j = D_j + W_j * Z_j$
     for $1 \le j \le \min(2, \frac{n}{k})$
     simultaneously.
```

d) The first column of B contains the solutions x_i for $1 \le i \le n$.

To justify this algorithm, we need only to prove the following claim:

Lemma 3 At the end of the i th iteration of Algorithm 1, we have

1) $\bar{y}_1^{(1)} = (x_1, x_2, \ldots, x_k)^t$;

2) for each partition

$$Y^{(i+1)} = \begin{bmatrix} y_{ik+1,ik} & & & \\ y_{ik+2,ik} & y_{ik+2,ik+1} & & \\ \vdots & \vdots & \ddots & \\ y_{ik+k,ik} & y_{ik+k,ik+1} & \cdots & y_{ik+k,ik+k-1} \end{bmatrix}$$

for $1 \leq i \leq \frac{n}{k} - 1$,

where y_{ij} are elements of the associated matrix Y of A; and

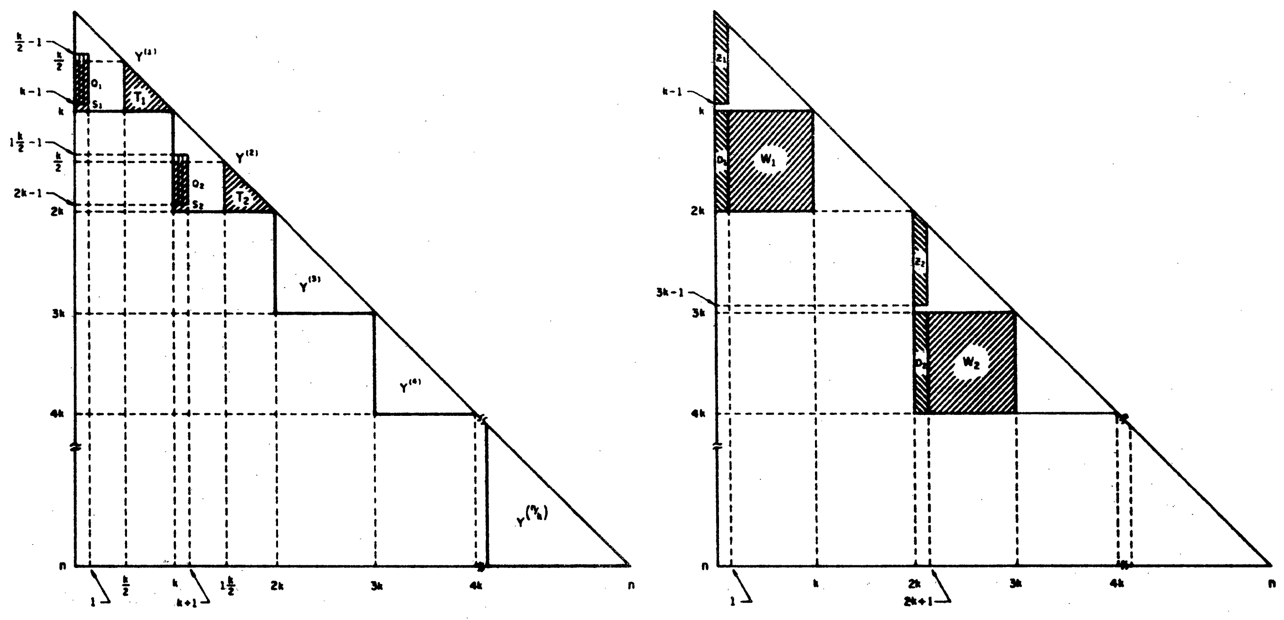
3) for $k \leq \frac{n}{2}$, D_1 is the vector of the partial sums of the first k terms in the expression of $x_{k+1}, x_{k+2}, \ldots, x_{2k}$, and similarly D_2 is that of $y_{3k+1,2k}, y_{3k+2,2k}, \ldots, y_{4k,2k}$ for $k \leq \frac{n}{4}$.

<u>Proof:</u> We prove this claim by induction on k. It is true for k = 2, as a basis. Let us assume it is also true for all iterations up to $\frac{k}{2}$ for some k as shown in Figure 1. Then, during the current iteration, by hypothesis condition (1) the first element of Q_1 is $x_{\frac{k}{2}}$ and S_1 is the vector of the partial sums of the first $\frac{k}{2}$ terms in the expression of $x_{\frac{k}{2}+1}$, $x_{\frac{k}{2}+2}$, ..., x_k by hypothesis condition (3).

Hence, we can write

$$\begin{bmatrix} x_{\frac{k}{2}} \\ \hline x_{\frac{k}{2}+1} \\ x_{\frac{k}{2}+2} \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} x_{\frac{k}{2}} \\ \hline S_1 \end{bmatrix} + \left[\begin{array}{cccc|c} 0 & & & & \\ \hline a_{\frac{k}{2}+1,\frac{k}{2}} & 0 & & & \\ a_{\frac{k}{2}+2,\frac{k}{2}} & a_{\frac{k}{2}+2,\frac{k}{2}+1} & 0 & & \\ \vdots & \vdots & \ddots & \ddots & \\ a_{k,\frac{k}{2}} & a_{k,\frac{k}{2}} & \cdots & a_{k,k-1} & 0 \end{array}\right] + \begin{bmatrix} x_{\frac{k}{2}} \\ x_{\frac{k}{2}+1} \\ x_{\frac{k}{2}+2} \\ \vdots \\ \hline x_k \end{bmatrix}$$

By virtue of Lemma 1 and hypothesis condition (2), the bottom partition of the above equation is equivalent to $(S_1 + T_1 * Q_1)$. This proves condition (1). Similarly, it can be shown that the matrix operation $S_2 = S_2 + T_2 * Q_2$ specified in step (iii) generates the results of $(y_{\frac{3}{2}k+1,k}$, $y_{\frac{3}{2}k+2,k}$, ..., $y_{2k,k})$, which combines with $(y_{k+1,k}, y_{k+2,k}, \ldots, y_{\frac{3}{2}k,k})$ from the previous iteration (hypothesis condition (2)) in forming the first column $\overline{y}_1^{(2)}$ of partition $Y^{(2)}$ of the current iteration.



(a) MATRIX B (b) MATRIX C

Figure 1. Dynamic Partitioning for Algorithm 1

With the new result of $\bar{y}_1^{(2)}$, now, we can see that shifting operation in step (iv) will result in condition (2) due to Lemma 2.

After step (iii) and step (iv), we know that $Z_1 = (x_1, x_2, \ldots, x_{k-1})^t$. Since in all previous iterations, no operation has been done on the data below the partitions on the diagonal, $Y^{(i)}$ for $1 \leq i \leq \frac{n}{k}$, we have

$$D_1 = (c_{k+1}, c_{k+2}, \ldots, c_k)^t ,$$

$$W_1 = \begin{bmatrix} a_{k+1,1} & a_{k+1,2} & \cdots & a_{k+1,k-1} \\ a_{k+2,1} & a_{k+2,2} & \cdots & a_{k+2,k-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ a_{2k,1} & a_{2k,2} & \cdots & a_{2k,k-1} \end{bmatrix} .$$

Hence, the resultant vector D_1 of step (v) is a vector of the partial sums as stated in condition (3). Similarly, it is true for D_2. This completes our proof.

Q.E.D.

It is worthwhile to note that the shifting operation stated in step (iv) could be easily accomplished by a proper indexing function, thereby no major computation time is involved and more efficient use of storage is possible by such a scheme.

To illustrate this algorithm, we use a R<8> as an example shown in Figure 2.

With this algorithm, we can now establish a general theorem about this class of problems.

Theorem 1. Any R<n> can be computed in T_p steps where $T_p \leq \log_2^2 n + 2\log_2 n - 1$.

The minimum speedup is $O(\frac{n^2}{\log_2^2 n})$ and the maximum number of processors required is $O(\frac{n^2}{4})$ for large n.

Proof: During the ith iteration of Algorithm 1, if a sufficient number of processors are given, then the matrix operation in step (iii) takes at most one multiplication time plus the time for the summation of at most $(\frac{k}{2}+1)$ operands, which is $\log_2 k$ addition steps. So that for a total of $\log_2 n$ iterations, this step alone takes

$$t_1 \leq \sum_{j=1}^{\log_2 n} j + \log_2 n = \frac{1}{2}\log_2^2 n + \frac{3}{2}\log_2 n.$$

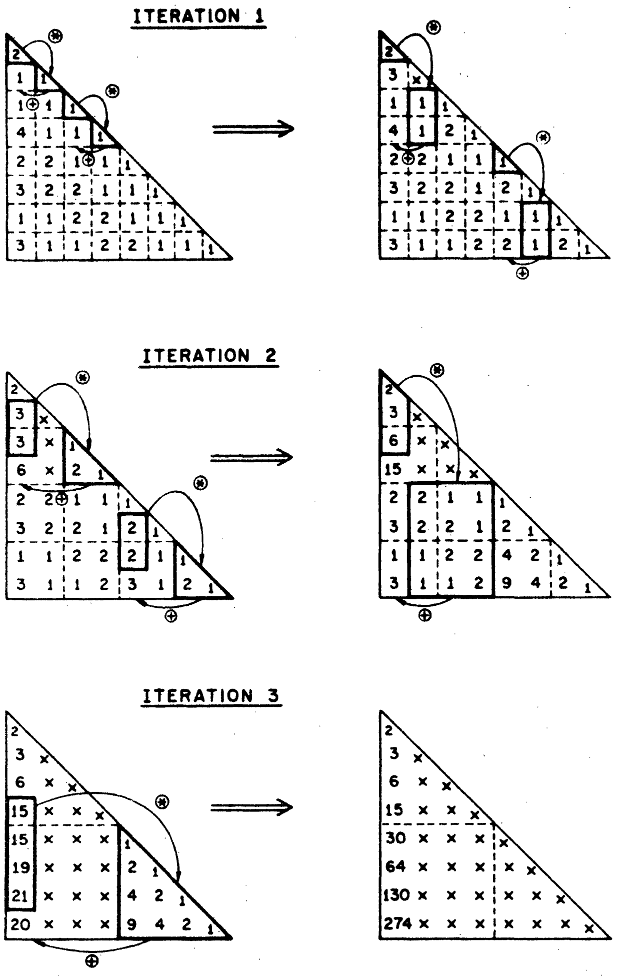


Figure 2. Parallel Evaluation of R<8>

Similarly, step (v) takes at most one multiplication time plus the time for the summation of k operands which is $\log_2 k$ addition steps. Thus, for a total of $\log_2(\frac{n}{2})$ iterations, this step needs

$$t_2 \leq \sum_{j=1}^{\log_2(\frac{n}{2})} j + \log_2 (\frac{n}{2}) = \frac{1}{2}\log_2^2 n + \frac{1}{2}\log_2 n - 1.$$

Therefore, the total computation time

$T_p \leq t_1 + t_2$, i.e., $T_p \leq \log_2^2 n + 2\log_2 n - 1$.

Since the serial computation time is n(n-1), the speedup S_p is

$$S_p \geq \frac{n(n-1)}{\log_2^2 n + 2\log_2 n - 1} = O(\frac{n^2}{\log_2^2 n}) .$$

To achieve the above speed, a sufficient number of processors should be provided at the multiplication time for each iteration i. So we will choose the maximum of the number of multiplications in step (iii) and step (v) as the number of processors required for that particular iteration. For step (iii), we can see from Figure 1(a) that

$$P_1(k=2^i) \le 2 \sum_{j=1}^{k/2} j \qquad \text{for } 2 \le k \le \frac{n}{2} ,$$
$$\le \sum_{j=1}^{k/2} j \qquad \text{for } k = n \quad .$$

For step (v), it can be observed from Figure 1(b) that

$$P_2(k) \le 2k(k-1) \qquad \text{for } 2 \le k \le \frac{n}{4} ,$$
$$\le k(k-1) \qquad \text{for } k = \frac{n}{2} \quad .$$

For large n, $p_1(n) \le (n^2 + 2n)/8$ and $p_2(n/2) \le (n^2 - 2n)/4$ give the respective peak values. Hence, the maximum number of processors is $0(\frac{n^2}{4})$.

Q.E.D.

As the basic algorithm stands, only $2\log_2 n-1$ unit time steps are multiplications, the rest being additions. Also note that in the above processor count, all processors perform uniform operations at each time step, and hence the bound can be further improved by lifting this restriction.

We close this section with some observations about the computational efficiency of this algorithm. Although its speed is approximately one-half that of Algorithm 1 in [1], the number of processors is reduced by a factor of n. Thus, it gives us a tremendous increase in computational efficiency. On the other hand, since R<n> is a special case of arbitrary coefficient system, we can use the latter algorithm to compute R<n> with some operations masked off [10]. In this case, it can be shown that the maximum number of processors required is $0(\frac{n^3}{128})$.

3. M^th Order Linear Recurrences with Constant Coefficients

In this section, we will turn to the considerations of the most practical m^{th} order recurrence problems, i.e.,

$$R\langle n,m\rangle: \quad x_i = 0 \qquad \text{for } i \le 0 ,$$
$$= c_i + \sum_{j=1}^{m} \alpha_j x_{i-j} \qquad \text{for } 1 \le i \le n ,$$

where $1 \le m < n$. In matrix notation $\bar{x} = \bar{c} + A\bar{x}$ with A being strictly lower triangular matrix of bandwidth m. For simplicity, we will assume both n and m are powers of 2.

Since A is now a band matrix, we can further speed up the computation of Algorithm 1 when $i \ge \log_2 2m$. This is done by changing partitions S_j, Q_j, T_j, Z_j, D_j and W_j in Figure 1 to that in Figure 3, and introducing new partitions U_j, V_j, G_j and E_j and their corresponding matrix operations as described in Algorithm 2. The proof of its validity can be found in Lemma 3 of [1].

<u>Algorithm 2</u>

```
a) Let B be a lower triangular matrix of order (n×n) in which
      $\bar{b}_1 = (c_1, c_2, \ldots, c_n)^t$ ,
      $\bar{b}_2 = (0, \alpha_1, \alpha_2, \ldots, \alpha_m, 0, \ldots 0)^t$,
   and $\bar{b}_j = \bar{b}_2[j-2]$, $3 \le j \le n$ .
b) Let C be an alias for B, i.e., B and C represent the same memory locations.
c) Repeat this step for i = 1, 2, ..., $\log_2 n$ ;
      (i)   Set $k = 2^i$ ;
      (ii)  Partition B and C as shown in Figure 1
            ($1 \le i \le \log_2 m$) and Figure 3
            ($\log_2 2m \le i \le \log_2 n$) .
      (iii) If $1 \le i \le \log_2 m$, then compute
            $S_j = S_j + T_j * Q_j$ for
            $1 \le j \le \min(2,\frac{n}{m})$ simultaneously;
            else compute $S_j = S_j + T_j * Q_j$
            for $1 \le j \le \min(2,\frac{n}{k})$ , and
            $V_j = V_j + T_2 * U_j$ for $2 \le j \le \frac{n}{k}$
            simultaneously.
      (iv)  Set $\bar{y}_j^{(i)} = \bar{y}_1^{(2)}[j-1]$ for
            $1 \le j \le k$, $2 \le i \le \frac{n}{k}$ ;
      (v)   If $1 \le i \le \log_2 m$ , then compute
            $D_j = D_j + W_j * Z_j$ for
            $1 \le j \le \min(2,\frac{n}{2m})$ simultaneously;
            else compute $D_j = D_j + W_j * Z_j$
            for $1 \le j \le \min(2,\frac{n}{2k})$ and
            $E_j = E_j + W_2 * G_j$ for $2 \le j \le \frac{n}{2k}$
            simultaneously.
d) The first column of B contains the solutions $x_i$ for $1 \le i \le n$.
```

For illustration, an example of R<16,2> is shown in Figure 4. In that figure, all numbers are kept in place to help understanding, although they may not be used after certain iterations.

From the above, we can obtain a general theorem for the m^{th} order linear recurrence problems.

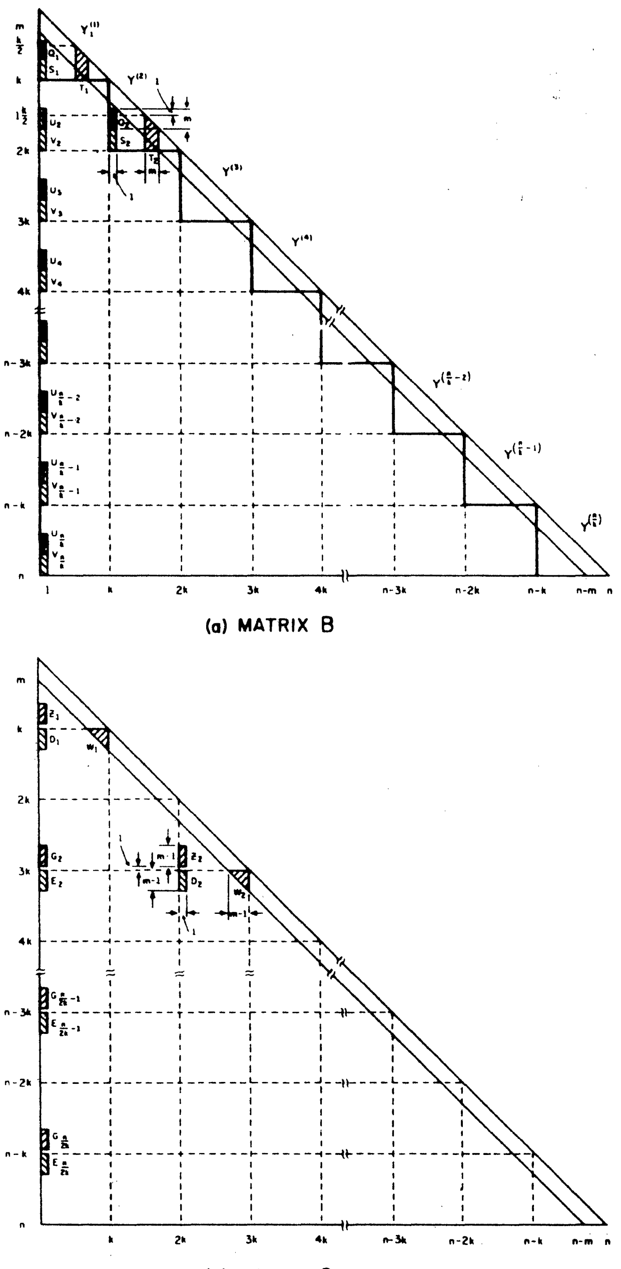


Figure 3. Dynamic Partitioning for Algorithm 2

Theorem 2. Any R<n,m> can be computed in T_p steps where $T_p \le 2\log_2 n$ for m = 1,

$\le (2\log_2 m + 3)\log_2 n - (\log_2^2 m + \log_2 m + 1)$ for m > 1 .

For n>>m, the minimum speedup is $O(\frac{mn}{\log_2 m \log_2 n})$, and the maximum number of processors required is O(mn).

Proof. For $1 \le i \le \log_2 m$, step (iii) is similar to that of Algorithm 1 and takes totally

$$t_1 \le \sum_{j=1}^{\log_2 m} j + \log_2 m = \frac{1}{2}\log_2^2 m + \frac{3}{2}\log_2 m .$$

However, for $\log_2 2m \le i \le \log_2 n$, this step needs only one multiplication time and one summation time of (m+1) operands per iteration. This amounts to

$$t_2 \le (\log_2(m+1) + 1)\log_2(\frac{n}{m})$$
$$\le (\log_2 m + 2)\log_2(\frac{n}{m}).$$

For step (v), it is similar to Algorithm 1 for $1 \le i \le \log_2(\frac{m}{2})$ and the total time is

$$t_3 \le \sum_{j=1}^{\log_2(\frac{m}{2})} j + \log_2(\frac{m}{2}) = \frac{1}{2}\log_2^2 m + \frac{1}{2}\log_2 m - 1 .$$

This same step will take one multiplication time and one summation time of m operands per iteration for $\log_2 m \le i \le \log(\frac{n}{2})$. Thus, in total it has

$$t_4 \le (\log_2 m + 1)\log_2(\frac{n}{m}) .$$

Hence,

$$T_p \le \sum_{i=1}^{4} t_i = (2\log_2 m + 3)\log_2 n - (\log_2^2 m + \log_2 m + 1),\ m > 1 .$$

For m = 1, since the W_j's diminish, $T_p \le t_1 + t_2 = 2\log_2 n$.

Comparing this with the serial computation time 2mn - m(m + 1), the speedup is

$$S_p \ge \frac{2mn - m(m+1)}{(2\log_2 m + 3)\log_2 n - (\log_2^2 m + \log_2 m + 1)}$$
$$= O(\frac{mn}{\log_2 m \log_2 n}) \qquad \text{for } n>>m .$$

In obtaining the above speed, we assume that there are enough processors at the multiplication time of any iteration. The number of multiplications required in step (iii) can be observed from Figure 1(a) and Figure 3(a) that

$$p_1(k = 2^i) \le 2\sum_{j=1}^{k/2} j \qquad \text{for } 2 \le k \le m ,$$
$$\le (\frac{n}{k} + 1)[\sum_{j=1}^{m} j + m(\frac{k}{2} - m)] \quad \text{for } 2m \le k \le \frac{n}{2} ,$$
$$\le \sum_{j=1}^{m} j + m(\frac{k}{2} - m) \quad \text{for } k = n.$$

For step (v), we can see from Figure 1(b) and Figure 3(b) that

$$p_2(k) \le 2k(k-1) \qquad \text{for } 2 \le k \le \frac{m}{2} ,$$

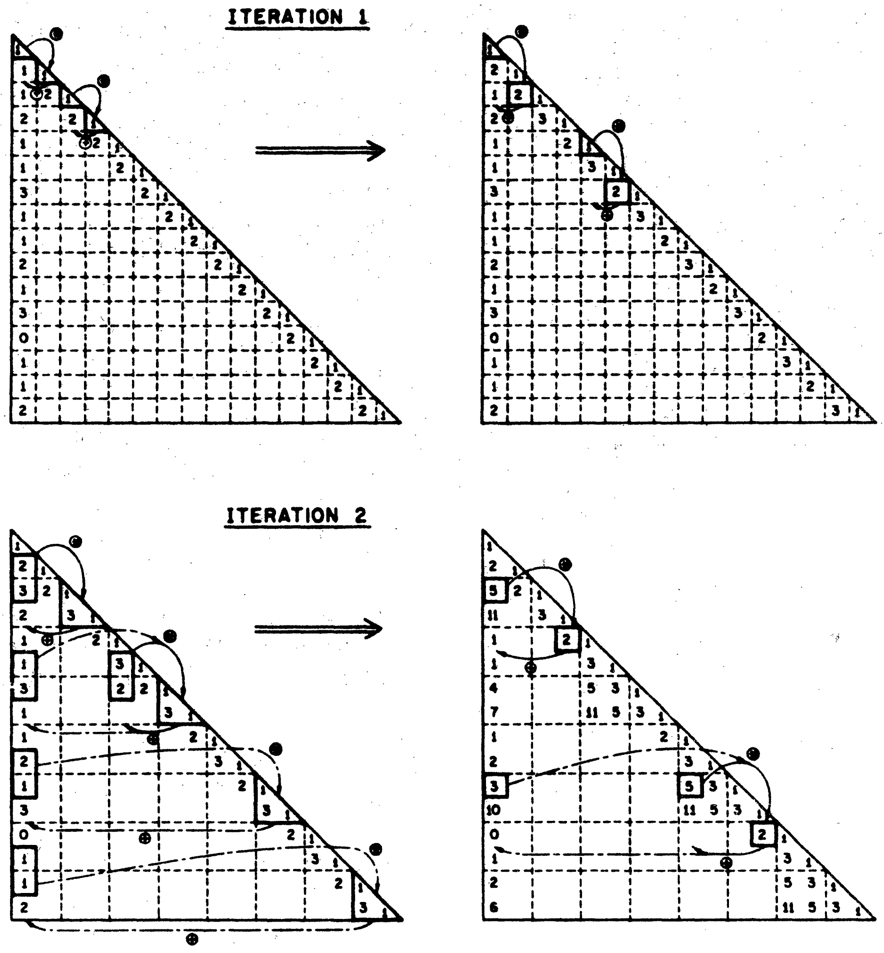


Figure 4. Parallel Evaluation of R<16,2>

$$\leq \sum_{j=1}^{m-1} \quad \text{for } k = m ,$$

$$\leq \left(\frac{n}{2k} + 1\right)\left(\sum_{j=1}^{m-1} j\right) \quad \text{for } 2m \leq k \leq \frac{n}{4} ,$$

$$\leq \sum_{j=1}^{m-1} j \quad \text{for } k = \frac{n}{2} .$$

When $n >> m$, $p_1(\frac{n}{2}) = (3mn - 6m^2 + 6m)/4$ is the maximum value and $p_2(k) = O(m^2)$ for all k's. Therefore, the processor bound is $O(mn)$.

Q.E.D.

As was true with Theorem 1, Theorem 2 requires only $2\log_2 n - 1$ steps. All processors are performing the same operations at the same time. Without this restriction, a better processor bound could be achieved. Also note that as m increases, greater efficiency can be achieved by using this algorithm instead of Algorithm 2 for arbitrary coefficients as described in [1]. On the contrary, we can achieve twice this speed by applying the latter algorithm with some operations masked off. In this case, the processor bound can be shown to be $O(mn + m^3)$, and one might prefer it as m becomes very small.

We will conclude this section with some comparisons between these results and some known algorithms in the evaluations of two distinct special cases. First, in computing the remote terms, e.g., x_n, in a homogeneous recurrence sequence

$$x_i + a_1 x_{i-1} + a_2 x_{i-2} + \ldots + a_m x_{i-m} = 0 ,$$

Miller and Brown [11] have shown an algorithm with about the same speed as that of Algorithm 2 by using $O(m^3)$ parallel processors. However, it can be shown easily that all vectors U_j, V_j, G_j and E_j are zero vectors, and hence those operations can be masked off totally. In addition, since we can leave out all unnecessary intermediate results, parts of the matrix operation $S_j = S_j + T_j * Q_j$ in step (iii) and $D_2 = D_2 + W_2 * Z_2$

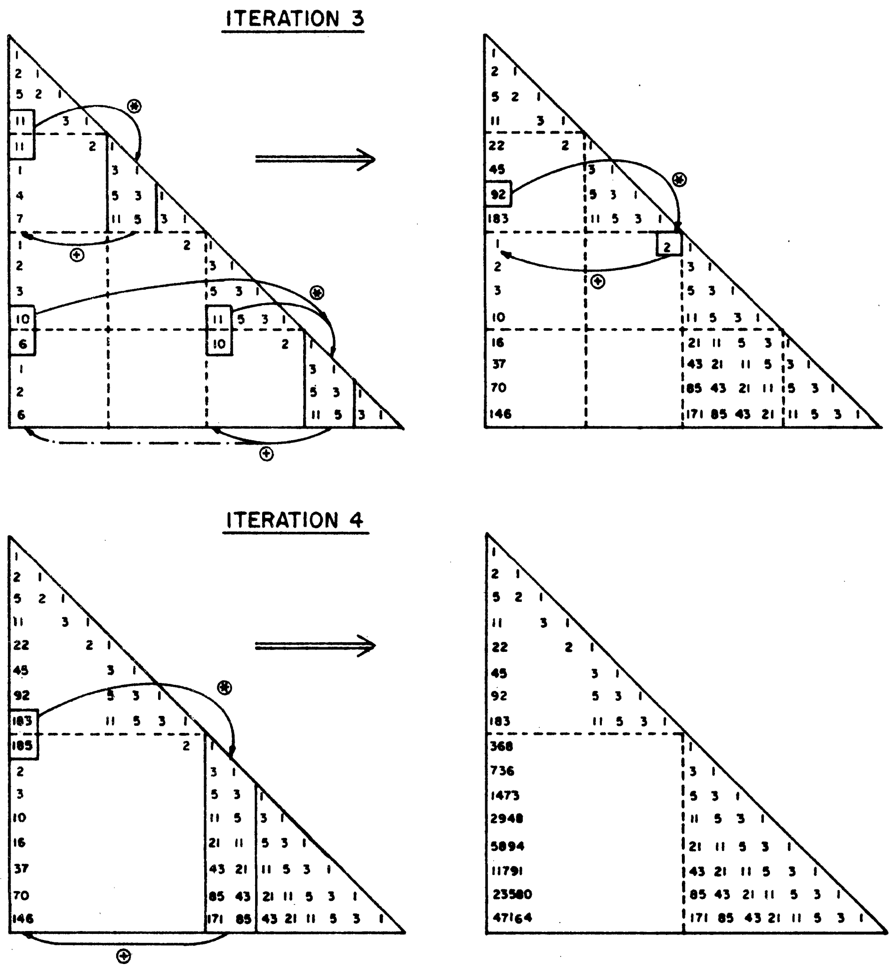


Figure 4. Parallel Evaluation of R<16,2> (Cont)

in step (v) can be further masked out and modified as shown in the new partitioning in Figure 5. Therefore, the maximum number of processors is $O(m^2)$. On the other hand, given $O(m^3)$ processors, we can use Algorithm 2 in [1] with the new partitioning as shown in Figure 6, and achieve twice the speed of the Miller-Brown algorithm.

As a result, we can establish the following facts.

Corollary 2.1. Any remote term x_n of a homogeneous linear recurrence relation

$x_i + a_1 x_{i-1} + a_2 x_{i-2} + \dots + a_{i-m} x_m = 0$,

can be computed within $(2\log_2 m + 3) \log_2 n$ time steps with $O(m^2)$ processors, and within $(\log_2 m + 2) \log_2 n$ steps with $O(m^3)$ processors.

Any polynomial $P_n(\alpha)$ of degree n can be represented as the last solution of the following linear recurrence relation

$R<n,1>$: $x_i = c_i + \alpha x_{i-1}$ for $1 \leq i \leq n + 1$.

By Theorem 2, this can be computed within $2\log_2 (n + 1)$ steps. However the processor bound can be reduced as we only need the last solution. This is done by changing the partitions T_1, Q_2, S_2 and T_2 in Figure 3 to that in Figure 5 (note that other partitions U_j, V_j in Figure 3 still remain). Thus, it can be shown to have the maximum demand of processors for k=2, i.e., $p \leq \left\lceil \frac{n+1}{2} \right\rceil + 1$. For illustration, let us evaluate $P_7(\alpha)$ by Algorithm 2 but with the modified partitions mentioned above on the matrix $B = [b_{ij}]_{8\times 8}$ of Figure 3. The whole process can be summarized as follows.

First, we compute

$$\alpha^2 = \alpha \cdot \alpha \ ,$$

and $b_{i,1} = c_i + c_{i-1}\alpha$ for $i = 2, 4, 6, 8$

simultaneously; then compute

$$\alpha^4 = \alpha^2 \cdot \alpha^2 \ ,$$

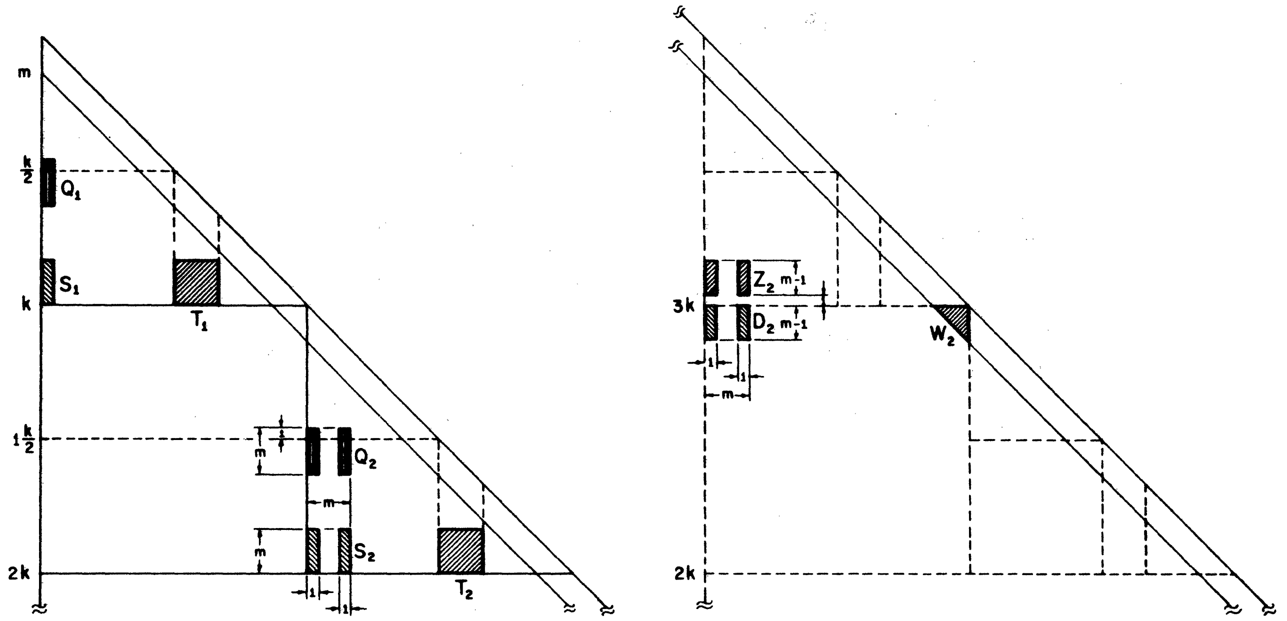


Figure 5. Dynamic Partitioning for Parallel Evaluation of Remote Term in Homogeneous R<n,m> by Algorithm 2

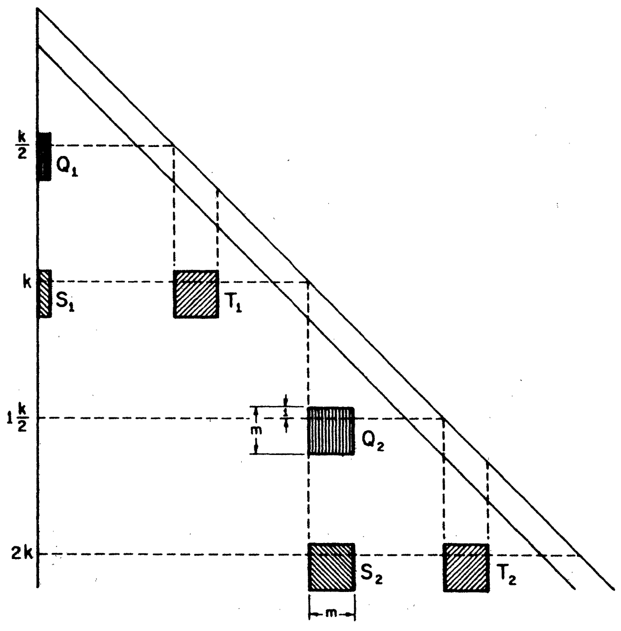


Figure 6. Dynamic Partitioning for Parallel Evaluation of Remote Term in Homogeneous R<n,m> by Algorithm 2 of [1]

and $b_{i,1} = b_{i,1} + b_{i-2,1}\,\alpha^2$ for i = 4, 8 simultaneously; and finally compute

$$b_{8,1} = b_{8,1} + b_{4,1}\,\alpha^4$$

which is now

$$= [(c_8 + c_7\alpha) + (c_6 + c_5\alpha)\alpha^2] + [(c_4 + c_3\alpha) + (c_2 + c_1\alpha)\alpha^2]\alpha^4 .$$

The final result in b_{81} is obviously the value of $P_7(\alpha)$. This procedure may be generalized to any degree n and the computational process will proceed like

$$\begin{aligned}P_n(a) = {} & c_n + c_{n-1}\alpha + (c_{n-2} + c_{n-3}\alpha)\alpha^2 \\ & + (c_{n-4} + c_{n-5}\alpha + (c_{n-6} + c_{n-7}\alpha)\alpha^2)\alpha^4 \\ & + (c_{n-8} + c_{n-9}\alpha + (c_{n-10} + c_{n-11}\alpha)\alpha^2 \\ & \quad + (c_{n-12} + c_{n-13}\alpha + (c_{n-14} \\ & \quad + c_{n-15}\alpha)\alpha^2)\alpha^4)\alpha^8 \\ & + \dots .\end{aligned}$$

It is interesting to note that the above procedure is exactly equivalent to the well-known Estrin's method [12]. In other words, Estrin's method becomes a very special case of our algorithms for evaluation of general R<n,m> systems. Hence, we can formalize it as a corollary.

<u>Corollary 2.2</u> Any nth degree polynomial can be computed within $2 \log_2(n + 1)$ time steps using at most $\left\lceil\frac{n+1}{2}\right\rceil + 1$ processors.

By comparing this result with the known k^{th} order Horner's rule, it is slightly faster and generally requires less number of processors to achieve the same speed [5]. Kuck and Maruyama [3] show that a general polynomial form of degree n can be evaluated in $O(\log_2 n + \sqrt{8\log_2 n})$ steps. While the speed is faster, it requires far more processors ($p = 2n$) than that required by our result. In practical applications where n is not very large, this method, even compared to the fastest known multifolding method [2], becomes attractive not only because of its simple implementation but also its easy integration with more general linear recurrence problems.

4. Practical Implications

The basic algorithms discussed previously are primarily for conceptual understanding. By more elaborate scheduling procedures or practical modifications of the original ones, one can develop more efficient computational algorithms which can use much less processors without significantly reducing the speedups.

For illustration, during the iteration at $k = \frac{n}{2}$ of Algorithm 1, we can perform the multiplications of each inner-product in step (v) within 2 steps instead of 1 step, thereby giving $p_2(n/2) = (n/2)(n/4) = n^2/8$ and hence changing the processor bound of Theorem 1 to $O(\frac{n^2}{8})$ and time bound to $\log_2^2 n + 2\log_2 n$. This folding scheme obviously halves the number of processors needed and almost retains the same speed as Theorem 1. Similar procedures can be applied to different iterations and extended to multiple-folding to achieve more efficient use of processors. Alternately, we can cut the triangular system into strips and then solve the recurrence system on the top of each strip with a sufficient number of processors. We then compute the partial sums in the bottom of that strip for the remaining equations as a new constant vector for the remaining triangular system. This process is repeated from the leftmost strip to the rightmost strip sequentially. This cutting scheme can also help in increasing efficiency as shown in [1].

We can conclude that a proper design and modification of the basic algorithms presented here for a particular environment should provide us high efficiency computations in exploiting future parallel machines.

Acknowledgment

I am grateful to Professor David Kuck for his stimulation and help. Professor Ahmed Sameh also provided fruitful discussions leading to this work.

References

[1] S.C. Chen and D.J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems," IEEE Transactions on Computers, July, 1975, pp. 701-717.

[2] K. Maruyama, "On the Parallel Evaluation of Polynomials," IEEE Transactions on Computers, Jan. 1973, pp. 2-5.

[3] D.J. Kuck and K. Maruyama, "Time Bounds on the Parallel Evaluation of Arithmetic Expressions," SIAM Journal of Computing, 4, 1974, pp. 147-162.

[4] R.H. Karp, R.E. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," Journal of the ACM, July 1967, pp. 563-590.

[5] Y. Muraoka, Parallelism Exposure and Exploitation in Programs, Department of Computer Science, University of Illinois, Report No. 424, Feb. 1971.

[6] P.M. Kogge and H.S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," IEEE Transactions on Computers, Aug. 1973, pp. 786-792.

[7] D.J. Kuck, Y. Muraoka and S.C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and Their Resulting Speed-Up," IEEE Transactions on Computers, Dec. 1972, pp. 1293-1310.

[8] D. Heller, "A Determinant Theorem with Applications to Parallel Algorithms," SIAM Journal of Numerical Analysis, 11, 1974, pp. 559-568; also, On the Efficient Computation of Recurrence Relations, Institute for Computer Applications in Sciences and Engineering (ICASE), June 1974.

[9] D.J. Kuck, "Multioperation Machine Computational Complexity," Proceedings of Symposium on Complexity of Sequential and Parallel Numerical Algorithms, Academic Press, 1973.

[10] S.C. Chen, Speedup of Iterative Programs in Multiprocessing Systems, Dept. of Computer Science, University of Illinois, Report No. 694, Jan. 1975.

[11] J.C.P. Miller and D.J.S. Brown, "An Algorithm for Evaluation of Remote Terms in a Linear Recurrence Relation," Computer Journal, Vol. 9, 1967, pp. 188-190.

[12] G. Estrin, "Organization of Computer Systems--the Fixed plus Variable Structure Computer," Proceedings of Western Joint Computer Conference, May 1960, pp. 33-40.

On the Ultimate Limitations of Parallel Processing

Jerome Rothstein
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210
614-422-7027

Summary Computation time depends on the problem, the computer, and how well they mesh. If we regard the computer as consisting of many interacting units the total time involves mutliples of fundamental computation times for units, inter-unit signal propagation times, the extent to which sequential operation of single units is replaced by parallel operation of many units, and the extent to which propagation delays and unit idle times can be reduced or eliminated. The first two contributions, reflecting fundamental physics and state-of-the-art device design, can be taken as imposed parameters. The last two, amenable to analysis by the methods of computer science (e.g., computability theory and computer architecture), are the chief concern of this paper. Specifically, a cellular automaton exerting local control on an iterative switching network (bus automaton, BA) is taken as a model of dispersed parallel computing capacity with communication between units. The communication paths (busses) are varied by the units to meet problem requirements. In the one-dimensional BA it is shown that the computation universality of the Turing machine is achieved along with parallel capability. Significant speed-up is demonstrated for the most general case. In many important cases the ultimate in speed-up is obtained. This is demonstrated explicitly for recognition of regular languages and other data processing executable by finite state machines. Other results are summarized and open questions discussed.

1. Introduction

Practical motivations for investigating the theory of parallel processing include the needs a) to decrease the time to solve a problem, b) to solve many problems simultaneously, and c) to increase the time sub-systems of a large system are actually computing. They are interrelated for satisfying one frequently contributes to others. They differ sufficiently, however, to permit different expedients to help in one case, but not in another. For example, faster devices and reduced propagation delays both help a), are less relevant to b), and may or may not contribute significantly to c). Parallel processing is not only relevant to all three, but one can assert that it is the only visible means to speed up computation for which insurmountable barriers set by physical limitations inherent in real devices can be by-passed.

This paper seeks to formulate problems in a manner permitting them to be attacked "all over" simultaneously, rather than sequentially and "locally", and how to utilize large parallel capacity to implement it. The cellular automaton (CA) is taken as the model of dispersed parallel capacity, and the equivalent of an iterative switching network controlled by the cells, which permits setting up communication busses between distant cells, enables them to cooperate. The resulting bus automaton (BA) is shown even in the one-dimensional case, to have both the computation universality of the Turing machine, and impressive parallel capability. The proof utilizes a potentially infinite shift register to simulate the Turing machine, with addition replaced by an arbitrary binary operation (groupoid). Generalization to higher dimensionality extends parallel capability in the sense that it becomes easy to reach the "ultimate" for some problems in which difficulty is encountered in the one dimensional case.

The next section treats two main topics. First, we justify encapsulating the constraints imposed by the nature of space, time, physical devices and the interactions between them into two parameters, C_p and C_s. They can be viewed as unit time-costs associated with signal propagation and device state changes respectively. We then show their utility for BA theory, and that no essential generality is lost by restricting the discussion to homogeneous BA's with C_p and C_s taken as constants. Later discussion is "pure computer science"; C_p and C_s suffice to represent engineering reality.

Section 3 introduces the groupoid formalism and its realization by one dimensional CA's or shift registers, and demonstrates its computation universality even with its potential parallel capability "crippled".

In Section 4, the origin of the basic BA concept from the foregoing is sketched, and computations by finite state machines are shown to be "immediate", i.e., the ultimate in speed-up is actually achieved when the parallel capability is used. The proof is a special case of a general groupoid technique applicable to other problems. Some other immediate cases are discussed. A significant speed-up result is obtained for the most general computation.

Section 5 briefly relates BA research to some other parallelism studies and tries to assess its future prospects.

2. Physical Limitations on Computation Speed

For localized computing units the fundamental time-cost parameter is average time to change state, C_s. The computation time τ is proportional to the total number N of state changes required

$$\tau = NC_s \qquad (2.1)$$

For interacting units there is a communication time

$$\tau_{ij} = d_{ij}C_p \qquad (2.2)$$

where τ_{ij} is the time to propagate information between units i and j, d_{ij} measures their separation in a convenient distance unit, and C_p is the time-cost per unit of signal propagation.

Clearly N is non-physical in that it depends only on the problem, the program, and the logical (as distinguished from physical) organization of the unit. In contrast C_s is physical, reflecting the state of the art of device design and the necessity to "manufacture" a new state given the preceding one (and input).

For reliable device function there must be sufficient stability (high energy barrier) to prevent random spontaneous transition between states. This is expressed by

$$E_B >> kT \qquad (2.3)$$

Here E_B is the "height" of the energy "hill" to be surmounted by perturbations (e.g., noise, thermal fluctuations) to permit state change, k is Boltzmann's constant, and T absolute temperature. The strategies of lowering T or raising E_B are well known respectively in the form of refrigeration (or cryogenic techniques) to lower noise level, and raising thresholds.

One must not confuse E_B and ΔE,

$$\Delta E = E_2 - E_1 \qquad (2.4)$$

where E_2 and E_1 are energies (or energy levels) associated with states 2 and 1 respectively; the states can be near the same (low) level with the hill E_B between them. For selectivity E_B should be high for noise and for all signals but the desired one. For high sensitivity ΔE should be small. A desired signal should be able to "tunnel" through E_B and surmount ΔE, the unit then relaxing to a new state. There may be three (or more) contributions to C_s, namely times to circumvent E_B, to put in enough energy to take care of ΔE by absorption or emission, and to relax (usually a dissipative process). The uncertainty principle of quantum mechanics puts an absolute limit on Δt (second contribution); the others are often instantaneous (tunneling) or non-existent (relaxation) for quantum jumps between atomic (or nuclear) states. We have

$$\Delta E \, \Delta t \gtrsim h \qquad (2.5)$$

where h is Planck's constant. It applies to absorption and emission (an elementary communication event involves emission, propagation and absorption) and to macroscopic objects, described by statistical averages or limits, for large numbers of atoms, of quantum descriptions.

Though E_B and ΔE are very different, for atomic systems interacting with heat reservoirs we rewrite (2.3) as

$$\Delta E = nkT \qquad (2.6)$$

where n>>1, n being a kind of stability index, as the condition that fluctuations do not quickly "wash out" distinctions stored as differences in occupation probabilities between two states. We obtain

$$\Delta t \gtrsim h/nkT \qquad (2.7)$$

We interpret this to say that physics implies the existence of minimum times for reliable changes of state. The higher the stability index, the shorter these times can be. The bound (2.7), with C_s in place of Δt, is negligable in practical cases. For n = 100 and T = 300°K, C_s is about 1.6×10^{-15} seconds.

Calculation of C_s for real devices is no doubt prohibitively difficult; due to multiple interaction and propagation events within them. The result will generally be very much larger than indicated by (2,7). However, C_s is a characteristic performance parameter which the computer scientist can take as given. Any unit can be viewed as having a characteristic C_s resulting from the C_s's, C_p's, d_{ij}'s and N's of its devices and microprograms. We can take C_s as constant, for performance is bounded by two uniform cases where C_s is taken as the largest and smallest of the values associated with the units.

Discussion of C_p is simpler. Relativity imposes an absolute upper bound (velocity of light) on signal propagation or material transport. For electrical and optical signaling the bound is frequently achieved, and when not, reduction is generally appreciably less than order of magnitude. Also, change in C_p can often be "absorbed" into change of unit for d_{ij}, as with "optical path length" (product or line integral of refractive index and length, convenient in optics).

The d_{ij} reflect spatial arrangement of units, and for units occupying finite volumes, represent compromises between close packing to reduce communication delays and sufficiently sparse packing to avoid crosstalk, excessive temperature rise from energy dissipation in devices, and to permit convenient servicing. Just as C_s could be bounded theoretically by constant C_s cases, so can C_p be bounded by constant C_p cases. Simple extension of the same reasoning shows that network performance can be bounded by performance of networks of identical units. For similar reasons the networks can be taken as uniformly arranged in space. This implies that the total number of units which can receive information from a given unit in time t or less is proportional to t^3 in space, t^2 in the plane, and t along a line.

We now rewrite (2.2) as

$$\tau_{ij} = N_{ij} \, C_p \qquad (2.8)$$

where C_p is now a constant which can be taken as the propagation time cost to traverse a single unit, and N_{ij} is an integer giving the number of units crossed in going from unit i to unit j.

The discussion thus leads to the BA, i.e., the CA with communication between separated cells, as a theoretical vehicle to study ultimate limitations of parallel computation, embodies the physics and engineering in two characteristic time costs, C_s and C_p, and leads to characterizing the time cost C of computations as

$$C = N_s \, C_s + N_p \, C_p \qquad (2.9)$$

where N_s and N_p are integers depending on problem, program, and computer organization.

If the cell diameter be d and c the velocity of light, then we often have

$$C_p \sim d/c \qquad (2.10)$$

It is frequently justified to take

$$C_p << C_s \qquad (2.11)$$

For example, if a device of diameter 10^{-3} cm changes state in 10^{-9} sec $= C_s$, $C_p = 3 \times 10^{-14}$ sec, i.e., $C_s \sim 3 \times 10^4 C_p$. For densely packed devices, a speed-of-light signal then propagates to about 10^{12} devices or more in the time required for one state change. Analogous considerations may apply to the nervous system; axonal propagation velocities are meters/sec, synaptic devices (junctions) are micron size or less, neuronal recovery times are milliseconds or less, and we have perhaps 10^{11} neurons packed in our skulls. Both for brains and BA's it is probably a good approximation to consider the major time cost to come from C_s. For computer networks, with transmission delays much larger than C_s for devices, (2.11) is apparently grossly in error. We say apparently because the appropriate C_s may be a turnaround time or time in a queue. For $C_p \sim C_s$ a device may do simple jobs itself as rapidly as it can with help.

A seemingly different approach to computing time bounds for computation by spatially distributed units D is given by Dertouzos [1]. His bound for $T_D(n)$, the time to compute n-argument functions, assuming a maximum speed of energy flow, a minimum detectable energy, and a maximum power transmission density, is proportional to $n^{1/3}$. But this is basically the same as the $n \alpha t^3$ result mentioned before (2.8) (take cube root of both sides). The minimum time to collect data from n similar units (certainly a lower bound for a computation) is achieved when they are densely packed in a sphere at whose center collection occurs. It is proportional to the radius, i.e., to $n^{1/3}$.

3. Groupoids, Cellular Automata, Shift Registers, and Turing Machines

Our original motivation for studying groupoids was the feeling that structure and pattern in nature evolved from nearest neighbor interactions at a molecular level [2,3].

A set $G = \{a_1, a_2, \ldots\}$ is a groupoid if it is closed under a binary operation. This is expressible as a mapping $G \times G \rightarrow G$, or, with the binary operation ("multiplication") indicated by $\otimes$

$$a_i \otimes a_j = a_k \qquad (3.1)$$

The gap between "local" algorithm embodied in groupoid multiplication and "global" pattern is bridged by building geometric structures from groupoid elements (strings) whose growth law (production of daughter strings) is determined by the local algorithm. More formally, for finite G we define groupoid strings as words (finite or infinite) using G as their alphabet. From the parent string

$$\ldots a_i a_{i+1} a_{i+2} \ldots$$

we form the daughter string

$$\ldots d_i d_{i+1} d_{i+2} \ldots$$

where

$$d_i = a_i \otimes a_{i+1} \qquad (3.2)$$

Plane displays of a starting string and successive generations of daughters have been studied as exemplars of algorithmic pattern generation and proposed for investigating development problems of biological structure by Rothstein [2,3].

Computation of daughter strings is performable in parallel on a one-dimensional cellular automaton (CA), essentially a "shift-register accumulator" whose "logic" embodies groupoid multiplication; see Figure 1.

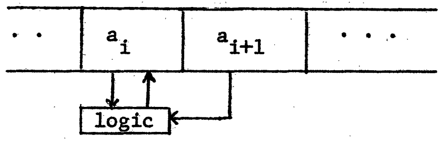


Figure 1. Shift Register Computation of Daughter String

The logic unit associated with cell i accepts inputs a_i and a_{i+1} from cells i and (i+1) respectively, substituting d_i for a_i in cell i. All units can clearly operate in parallel. This is a special CA; it is one-dimensional, and its neighborhood function has two arguments. For classical CA theory see von Neumann [4], Burks [5], Codd [6], Smith [7,8], Banks [8], Nourai and Kashef [10], and others. Generally, the next state of a cell is determined by the states of n neighbors. We now show that the groupoid formalism can represent strings over a symbol set closed under an n-ary operation by taking (n-1)-plets as groupoid elements. Successive daughter-string symbols are formed by the n-ary operation over the n contiguous symbols obtained from the previous set by dropping the leftmost symbol and adding the next symbol of the parent.

Consider the parent string

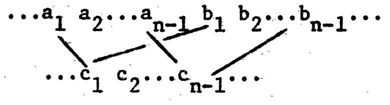
$$a_1 a_2 \ldots a_{n-1} b_1 b_2 \ldots b_{n-1}$$

which can be viewed as the concatenation of two (n-1)-plets. Let the daughter (n-1)-plet, $c_1 \ldots c_{n-1}$, be generated from contiguous parent (n-1)-plets, by the element-wise n-ary operation:

$$G^n \xrightarrow{g} G$$

$$\begin{aligned} c_1 &= g(a_1 \ldots a_{n-1} b_1) \\ c_2 &= g(a_2 \ldots a_{n-1} b_1 b_2) \qquad (3.3) \\ &\cdots\cdots \\ c_{n-1} &= g(a_{n-1} b_1 b_2 \ldots b_{n-1}) \end{aligned}$$

Clearly this process produces one (n-1)-plet from two given (n-1)-plets, thus defining a groupoid. Figure 2 illustrates the process graphically.



$\ldots a_1 \; a_2 \ldots a_{n-1} \; b_1 \; b_2 \ldots b_{n-1} \ldots$
$\ldots c_1 \; c_2 \ldots c_{n-1} \ldots$

Figure 2. Groupoid Daughter for n-ary Mapping

The groupoid mapping can be symbolized by the notation [2]→[1], an n-ary mapping by [n]→[1]. A one-dimensional CA (Figure 3), where the next state of A_i is determined by the present states of

A_{i-1}, A_i and A_{i+1}, thus defines a "state-string" and an n-ary state mapping with n=3.

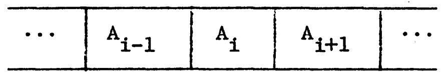


Figure 3. One Dimensional CA

Its "time-history" is thus expressible as a succession of daughter strings over a groupoid whose elements are doublets of automata states.

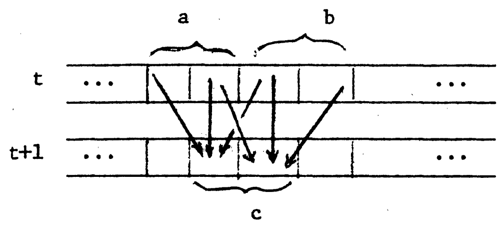


Figure 4. Parent at Time t, Daughter at Time (t+1)

Figure 4 shows this for contiguous doublets a and b of the (parent) string at time t, producing doublet c of the (daughter) string at time (t+1).

The above shows that the one dimensional CA is contained within the groupoid formalism. The Turing universality (computation universality) of the groupoid formalism will now be demonstrated by explicit construction of a groupoid simulating an arbitrary Turing machine (TM) using a one dimensional CA.

The TM has a finite state control and an infinite tape, ruled in "squares" containing symbols ("blank" is a symbol). The control is in one of a set of states K,

$$K = \{q_0, q_1, \ldots, q_r\} \tag{3.4}$$

when it scans that one of the set of tape symbols, Σ,

$$\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_s\} \tag{3.5}$$

at its current address (say n) on the tape. It then makes a transition to a new state, prints a new symbol at its address n, and moves right or left to address $n \pm 1$. General data processing, computation, or procedures are a succession of such steps. The initial symbol string on the tape is the input. It is usual to take q_0 for the initial state; with the initial address at a left end marker or first symbol of the initial string. When a state "halt" is entered, the machine stops. What is left on the tape is output, processed data, or the result of computation, or the halt state is taken as embodying the desired decision (reject or accept, etc.). The program is embodied in the transition function. More formally, the action of the machine is defined by a mapping

$$K \times \Sigma \to K \times \Sigma \times \{\text{right, left}\} \tag{3.6}$$

which specifies, for current state and input symbol, the next state, output symbol, and "move" to a new address. The mapping is often given as a list of quintuples or as a rectangular table of triplets (next state, output symbol, move), with rows and columns labelled by current state and input symbol.

To simulate TM by CA associate square i of TM with automaton A_i of CA, naming states $\{Q_j\}$ of the (identical) A_i by

$$\{Q_j\} = K \times \Sigma \times \{\text{on, off}\} \tag{3.7}$$

Cell A_i is "on" if and only if the TM control is scanning the symbol at address i; all other cells are "off" (quiescent). State transition rules mimic TM rules: the $K \times \Sigma$ part is unchanged, and {right, left} of TM corresponds to {on, off} of CA as shown in Figure 5.

1. (a) TM Control in q_r scanning σ_s at address i
   (b) A_i in state $(q_r, \sigma_s$, on)
2. (a) TM prints σ_t at address i, moves R in state q_u, scanning σ_v at (i+1)
   (b) A_i in state $(q_r, \sigma_t$, off), A_{i+1} in state $(q_u, \sigma_v$, on)
3. (a) TM prints σ_t at address i, moves L in state q_w, scanning σ_x at (i-1)
   (b) A_i in state $(q_r, \sigma_t$, off), A_{i-1} in state $(q_w, \sigma_t$, on)
4. (a) Symbols on squares j not currently the address of TM control are unchanged
   (b) σ-component of states of all A_j currently "off" are unchanged

Figure 5. CA Simulation of TM

Less formally, on or off is determined by L or R outputs of TM, namely L means $A_i \to$ off, $A_{i-1} \to$ on while R means $A_i \to$ off, $A_{i+1} \to$ on. The asserted simulation now follows whence TM is simulated also by a (unidirectional) "shift-register accumulator" and represented by the groupoid string daughter formalism over cell state doublets. The simulation amputated parallel capability but preserved universality, whence groupoids, CA's and BA's invite investigation as vehicles with general parallel computation capability. Daughter string computation is a paradigm for total parallelism. Given a daughter, finding a parent (or parents) or more remote "ancestors", may entail much sequentiality.

4. Development and Initial Applications of Bus Automata

The immediate stimulus to developing the BA concept was a refusal to accept parent string computation, given the daughter, as inherently sequential. The key idea used, however, developed from research on pattern recognition by retina-like devices, specifically straight line recognition by a plane CA (Rothstein and Weiman [11,12,13]). There string manipulations, incident to recognizing whether an encoded candidate configuration was a straight line or not, were enormously facilitated if cells could enter a conducting state, permitting an appropriate neighbor to effectively augment its set of neighbors by those of the conducting cell. A logic bus, permitting "broadcast" type communication between cells as a group and a "cell synchronizer" was also used to simplify automata design, timing problems, recognition algorithms, and to enhance parallelism. The logic bus was fixed, so essentially the entire capability to adapt communication paths to fit the problem was contained in the ability of cells to enter a conducting state. We show how this "conduction trick" permits parallel computation of a parent string.

If $\ldots p_i \ldots$ is a parent string of given daughter string $\ldots d_i \ldots$, then the possible candidates for p_i are all those elements g_i of G for which another element of G, say g_{i+1}, exists satisfying

$$g_i \otimes g_{i+1} = d_i \qquad (4.1)$$

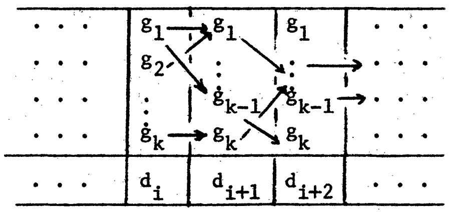


Figure 6. Parallel Computation of Parent Strings

In Figure 6, we display $\ldots d_i \ldots$ with its elements in boxes corresponding to CA cells, and with all the elements of G, in some standard order, written as a column $g_1, g_2, \ldots, g_k$ above each d_i. The arrows go from each element g_i above d_i to each element g_{i+1} above d_{i+1} satisfying (4.1); this is done for all addresses i. We thus have a directed graph whose vertices are labelled by elements of G and whose edges are determined by the groupoid multiplication table and the string $\ldots d_i \ldots$.

The set of possible parents is then represented by the set of all continuous chains of edges constructed above the string $\ldots d_i \ldots$. For G a quasigroup of order k (a quasigroup is a groupoid with unique two-sided solvability, i.e., for all a,b,c in G unique x and y exist such that $a \otimes x = c$ and $y \otimes b = c$) there are precisely k possible parents for any string. A "grandparent" string (or more remote ancestors) comes under the same discussion, for a granddaughter string is produced by a $[3] \rightarrow [1]$ mapping defined by G. It follows that telescoping k "generations" into one parallel computation can always be done with increased groupoid complexity (this expresses a well known time-complexity trade-off).

Viewing edges as conducting paths, with complete paths connecting an indicator lamp to a power source, say, and noting that the circuit logic to establish them is determined by the groupoid (and therefore a "setting" made when the data is put in), shows that parallelism is essentially complete. Also, specifying a single element p_i selects a unique chain (if it exists). As all logic is prewired the time cost to obtain a parent is that of one logical state setting (state change) C_s. The "read-out" time is nC_p, where n is word length and C_p propagation time cost for output signal to traverse one cell. Readout need not be counted as part of computation time, for the "answer" is already stored in the state of the system.

The foregoing applies verbatim to acceptance of a regular language. The groupoid operation is concatenation for the syntactic monoid or its generators. Initial and final elements of the string correspond to starting and accepting states. Regular languages and their relation to finite state automata are treated extensively in many texts, e.g., Chapter 3 in Hopcroft and Ullman [14]. Acceptance is "immediate" for only the time for a single state change is needed. Were none required there would be no problem, the input being the solution. More generally, if k successive state changes are needed for acceptance, where k is a constant <u>for all words in the language</u>, no matter how long, we still call acceptance immediate. The reason is that a k-tuplet groupoid can always be constructed for this case in which acceptance is immediate.

Though the transcription is trivial from regular language acceptance to arbitrary computation, translation, or data processing by finite state machines, we discuss it to bring out a strong point of the groupoid technique. It is that homomorphic mappings can be introduced with no additional complications. Refer to Figure 6, interpreting $\ldots d_i \ldots$ as the output string of a finite state automaton (FSA). For a Moore FSA outputs are additional labels on the states (i.e., a homomorphic mapping of the state set) and $\{g_1, \ldots g_k\}$ can be identified with the set of states of the FSA. The inputs are simply arrow labels, traditionally from the same alphabet as d_i for CA's but not so for finite state transducers and many other FSA's. For Mealy FSA's inputs and outputs are arrow labels, but no change is needed in Figure 6. However, the usual convention is that current state and input determine next state and output, i.e., in (4.1) d_i becomes d_{i+1}, an alternative which can always be used in place of (4.1) if desired. Also, as any edge is determined by the vertices it joins, and one vertex and an arrow determine the other vertex, pairs (g_i, g_{i+1}) correspond precisely to pairs (g_i, I_i), where I_i is the FSA input in state g_i causing the FSA to enter state g_{i+1}.

This now leads to a <u>general speed-up theorem for Turing machines</u>. As any TM constrained to move in one direction is simply an FSA whose inputs and outputs are on the tape, the foregoing implies that a one-dimensional BA can do in time C_s (immediately) what TM does <u>between</u> reversals on its tape. It switches between being a "left FSA" and a "right FSA" at each reversal. The BA thus cuts the time cost to

$$C = (r+1)C_s + \sum_{i=1}^{r} \ell_i C_p \qquad (4.2)$$

where r is the number of TM reversals and ℓ_i is the number of cells containing the substring processed "immediately" between the ith and the (i+1)th reversals. Propagation costs have been included because the substrings are effectively read out at each reversal. If propagation cost is neglected we have the (usually realistic) cost

$$C = (r+1)C_s \qquad (4.3)$$

Seemingly more impressive is the result that if there is a fixed upper bound on tape distance (number of cells) between reversals for a TM, a BA exists doing the calculations of that TM immediately. However, it follows easily from the foregoing and k-tuplet speedup, where k is now the upper bound. With a simple convention about null symbols, all substrings scanned between successive reversals can be regarded as k-tuplets (k is the maximum ℓ_i of (4.2)), each viewed as a single groupoid element and thus equivalent to one symbol. They are processed alternately by corresponding versions of R and L FSA's. But a single FSA is easily constructed to embody this alternating behavior whence the stated result follows.

To summarize, the one-dimensional BA, with communication along busses reduced to mere continuity check, in effect, achieves ultimate speed-up for finite state computations, is computation universal, and accomplishes potentially tremendous speed-up generally. When (4.3) is valid the ratio R of TM time cost to BA time cost for the same C_s is

$$R = (\sum_1^r \ell_i)/(r+1) \qquad (4.4)$$

As R is the number of squares visited by the TM control divided by one plus the number of turn-arounds, it is at least one (realized only in a trivial case), and has no finite upper bound in general.

Two (or higher) dimensional BA's are more powerful than one dimensional BA's for several reasons. They are easier to apply to geometrical or pattern problems. They permit setting up an unlimited number of busses parallel to a row of cells, or "detouring" around a region of cells. The number of cells to which propagation can occur increases quadratically, rather than linearly, with time (cubic increase for 3D BA's; higher dimensionality is "non-physical"). This implies non-existence of finite upper bounds on how much "more parallel" computations of a planar (or cubic) BA can be compared to those of a linear BA (or cubic compared to planar).

The first point has been illustrated both for straight line recognition and for determining topological connectivity of regions in the plane (Rothstein and Weiman, [11]). The latter is of special interest here, being particularly vexing with sequential approaches yet almost trivial for a plane BA. The second and third points are illustrated by the following ancedote. Several years after Weiman received his degree, Moshell sought a dissertation topic, so the writer suggested investigating parallel computation by BA's, specifically for acceptance of formal languages more general than regular. The "ultimate" result for regular languages and a powerful speed-up for the most general case had already been obtained on the one dimensional (1D) BA. Significant parallel speed-up for a context sensitive (CS) language (line codes [11]) had been obtained with a 2D BA, and the writer had also found how to speed up acceptance of Dyck languages, which are context free (CF), by the 1DBA, using only nearest neighbor conduction, and a number of computation steps equal to the depth of the "deepest nest" in the string (no limit on the number of nests). A bar to Dyck language immediacy on the 1DBA is the impossibility of threading an indefinitely large number of channels through an FSA. Moshell found this easy for the 2DBA to overcome, whereupon the writer showed that the same method worked for many context sensitive languages like $(a^n b^n c^n,,,t^n)$ and $((w\ w^R)^*)$, where w is an arbitrary word and w^R is w written in reverse order. We suspected, from the above and the first speed-up result, that regular languages, and only they, were immediate on the 1DBA. This turned out to be so, but for 2D and 3D CA's the results were less tidy. Indeed, a major part of Moshell's dissertation dealt with the problems of precisely characterizing immediate languages and their relations to known hierarchies of languages (with respect to generality, complexity, etc.).

The work of Rothstein and Moshell [15,a,b,c,d], both published and in preparation, has yielded some unexpected results. Complexity with respect to parallelism (immediate languages are simplest) is less for some CS languages, which are highly complex from a conventional viewpoint, than for general CF languages. For example, words over an alphabet of one letter whose length is a prime number form a CS language. From number theory one would expect it to be very complex, but it is immediate. The Cocke-Younger algorithm leads to a $(\log_2 n)C_s$ time bound for CF languages. We do not know if this can be improved.

5. Retrospect, Prospect and Concluding Remarks

The BA is both a CA and an iterative logic (microcellular) array. It is thus more powerful than either, andis microprogrammable in principle. Array research is summarized in Minnick [16]. Microprogrammed arrays have been discussed by Jump and Fritsche [17]. The BA is also a multiprocessor. Some theoretical aspects of multiprocessors have been surveyed by Baer [18], and statistical modelling of their performance considered by Sastry and Kain [19]. A discussion of several computer organizations and their effectiveness, including some parallel processor and multiprocessor aspects, is given by Flynn [20]. Many lines of parallel computation research are treated in a special IEEE Transaction issue [21], and in the proceedings of this conference and its predecessors. Kuck, Muraoka and Chen [22], analyzed Fortran-like programs at the statement level to find simultaneously executable operations.

The overwhelming mass of the literature, lightly sampled above, on array multiprocessor, content addressable parallel processors [23], and other parallelism research has either been in anticipation of technological advance (e.g., array research and LSI) or under pressure of handling enormous work loads in real computer environments (STARAN, etc.). Fundamental aspects and ultimate limitations were thus largely neglected in favor of practical ones. While many fundamental questions are treated in CA research, usual absence of fast communication between distant cells made their impact on parallel processing problems small.

We believe BA research has made and will make progress toward fundamental understanding of parallel processing, and that it will eventually contribute to operating system design for maximizing speed and throughput, to fully utilizing the potential of LSI and other technological advances, and to rational device and system architecture, where geometric arrangement, local and system logical design and communication bus systems merge into a powerful whole. We believe

BA research will have enormous impact on complex system modeling, the BA itself often becoming an analog of the system modeled. The straight line recognizer [11,12,13] did exactly this for a rudimentary visual pattern recognition system with a simple BA. Indeed, we now think of much of the formal side of theoretical science as design of BA's, of which systems of scientific interest are analogs. We expect this to recur for modeling "higher cognitive functions", adaptive systems, artificial intelligence, etc..

What can we say about ultimate limitations on parallel processing not soon to be proven wrong by future research? As in the text, there are physical and computer science aspects. Turning first to the physical, we expect a form of resolution of time cost into C_p and C_s components to be maintained, for the velocity of light to be the ultimate upper bound on signal transmission speed, and for that bound to be frequently achieved. Both C_s and unit size (and thus C_p normalized to unit size) are expected to shrink toward atomic space and time scales with future advances. Estimated values of C_s and C_p are likely to have short lived validity. The elementary computation act, like measurement, is irreversible, and the quantum statistical mechanics of irreversible processes is far too primitive to permit making the preceding statement much better. It seems clear, however, that much improvement in practical devices and systems can occur before ultimate C_s and C_p limits are encountered. Computation of those limits is a question for future research.

With C_s and C_p as given parameters, can computer science set ultimate bounds on time cost or speed-up for parallel computation? As shown in the text, (4.1) leads to the "ultimate" result for regular languages, later generalized to immediate languages [15,a,d] (which include many context free and context sensitive languages, but which have not been shown to properly include the context free ones). The result (4.2) gives explicit values for N_s and N_p of (2.9) in the most general case on the 1DBA but r and 1_i of (4.2) are complexity measures hard to compute, and vast improvements are possible for 2DBA or 3DBA. For language recognition problems, N_s and N_p are functions of n; languages can be hierarchically arranged in complexity according to the kinds of function involved. Immediate cases had N_s bounded by a constant, N_p by a linear function. We expect "parallel complexity theory" to be "two-dimensional" (s and p "components"); an important practical problem is how to optimize trade-off between components. But that may be the "easy" part because programs and generative grammars exist, of widely differing complexity, which respectively perform the same class of computations or define the same formal language. Worse yet, there is no way, in general, to tell if two programs perform the same class of computations or if two grammars generate the same langauge! But let us count our blessings: many interesting languages are immediate; the (s,p) bound for <u>all</u> context free languages is surely not higher than $(k_1 \log_2 n, k_2 n)$, where the k's are constant and n is word length [15d]; (4.2) can surely be improved; given any TM, there exists a faster TM performing the same class of computations. So optimism seems to be in order: no inherent general limitations short of immediacy have been shown to exist and tremendous practical speedup is surely possible.

<u>References</u>

[1] M. L. Dertouzos, IEEE Trans. Computers, <u>C-22</u>, 12-17, (1973).

[2] J. Rothstein, Patterns and Algorithms, Proc. 1970 IEEE Symp. Adaptive Proc. (9th), paper II.4 (1970). Available as Tech. Report.

[3] J. Rothstein, Algorithmic Pattern Generation as a Basis for Biological Structure, Biophysical Soc. Abstr. <u>10</u>, p. 233a (1970).

[4] J. von Neumann, Theory of Self-Reproducing Automata, University of Illinois Press,(1966).

[5] Essays on Cellular Automata, A. W. Burks, ed., Univ. of Illinois Press, Urbana, Illinois (1968).

[6] E. F. Codd, Cellular Automata, Academic (1968).

[7] A. R. Smith, J. Coptr. Sys. Sci. <u>6</u>, 233-253, (1972).

[8] A. R. Smith, Introduction and Survey of Polyautomata Theory, intro. to German translation of [4], Rogner and Bernhard GmbH., Munich (1975).

[9] E. R. Banks, IEEE 11th Ann. Symp. Switching and Automata Theory, pp. 194-215, (1970).

[10] F. Nourai and R. S. Kashef, IEEE Trans. Computer, <u>C-24</u>, pp. 766-776, (1975).

[11] J. Rothstein and C. F. R. Weiman, Computer Graphics and Image Processing <u>5</u>, 106-124, (1976).

[12] C. F. R. Weiman and J. Rothstein, 1975 Sagamore Conf. Parallel Proc. 168-170, (1975).

[13] C. F. R. Weiman and J. Rothstein, Pattern Recognition by Retina-Like Devices, Report, OSU-CISRC-TR-72-8 (AD 214 665/2), (1972).

[14] J. E. Hopcroft and J. D. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, Reading, Mass. (1969).

[15] In addition to papers of Moshell and Rothstein at this conference, see:
(a) Bus Automata, report CS-76-14, Dept. of Computer Science, U. of Tenn. (1976);
(b) Immediate Languages, report CS-76-16, Dept. of Computer Science, U. of Tenn. (1976);
(c) Bus Automata and Parallel Computation, Proc. 1976 Southeastern Symp. System Theory, U. of Tenn., Knoxville, pp. 246-252.
(d) J. M. Moshell, Parallel Recognition of Formal Languages by Cellular Automata, Ph.D dissertation The Ohio State University, Columbus, Ohio, 1975.

[16] R. Minnick, JACM <u>18</u>, 203-241, (1967).

[17] J. R. Jump and D. R. Fritsche, IEEE Trans. Computers, <u>Vol.</u> <u>C-21</u>, 974-984, (1972).

[18]J. L. Baer, Computing Surveys, <u>Vol. 5</u>, 31-80, (1973).

[19] K. V. Sastry and R. Y. Kain, IEEE Trans. Computers <u>Vol.</u> <u>C-24</u>, 1066-1074 (1976).

[20] M. J. Flynn, IEEE Trans. Computers, <u>Vol.</u> <u>C-21</u> 948-960 (1972).

[21] IEEE Trans. Computers, August 1973. Vol <u>C-23</u> #8

[22] D. Kuck, Y. Muraoka, S. Chen, IEEE Trans. Computers, <u>C-21</u>, 1293-1310, (1972).

[23] C. C. Foster, Content Addressable Parallel Processors, Van Nostrand, New York (1976).

AN EFFICIENT MULTIPROCESSOR ARCHITECTURE

by Vincent UNG
INSTITUT DE PROGRAMMATION
Université Pierre et Marie CURIE - PARIS -
(FRANCE)

Abstract : In a parallel/distributed processing system, the efficiency of such a system depends essentially on the manner in which the processors intercommunicate. In the standard way, communication uses the internal buses, the I/O buses or the DMA channels. In every case, the data transfer takes time. In a multiprocessor system, each processor generally has a specific task and possesses a unique structure such as its word length. As a general rule, transfer processing, managed by firmware or software, is needed to convert between different word lengths and to store the data.

This paper describes a hardwired method which facilitates both rapid communications between processors, and at the same time, rapid word length transformation, in a local multiprocessor system.

I - Introduction

Because of the proliferation of more and more sophisticated microprocessors and low cost minicomputers, distributed and parallel processing is more interested in decentralizing the data processing to decrease cost and increase efficiency. We can envision a multiprocessor in control of peripheral devices [1], syntactic filtering [2], lexical processing [3], arithmetic processing etc...

The processors have to inter-communicate [10], or the system is not a multiprocessor. As a general rule, communication is realized by using internal buses, input-output buses or DMA channels and reduces the efficiency of the whole system. The causes for the reduced efficiency are.

- Data strangling on the buses.
- transfer time that causes waiting
- transfer processing as the word length transformation and the storage function.

II - Multiprocessor justification

Our reseach on an APL machine which incorporates automatic evaluation of calculation errors caused by truncation [4] led us to propose a multiprocessor architecture rendered necessary by two aspects of APL.

- conversation
- Nümerical treatment.

The conflict between these aspects is clear: the first is essentially slow and the second needs a high speed treatment. One approach to resolve the problem is a two-processor system :

- An eight-bit microprocessor [5] which treats a string of symbols coming from the APL terminal, codes it in internal codes, processes the syntactical analysis, creates executable data in a common memory block for the second processor, and then delivers the results for the terminal.
- A thirty-two- bit high-speed processor based on bipolar technology [6,7] which is microprogrammed for numerical calculations : array treatment, floating point arithmetic, and precision evaluation.

III - Communication.

In the usual mode, the drawbacks are evident. In this paper, we present a hardwired method permitting the rapid transfer of a memory block and at the same time, rapid word length transformation (fig.1).

With this method, we have true parallel processing : two processors work simultaneaously without the drawbacks of shared buses.

IV - Memory organization

Around the memory and for each processor, we build a data bus and an address bus, corresponding to the structure of this processor. An exemple illustrates this organization : the memory is formed out of 1K-bits RAM chips. Each 8 chips forms one indivisible sub-block of 1K-8 bits. This size is determined by the smallest processor (8 bits). The 32 bit processor will determine the minimum number of sub-blocks per block. In this case, four sub-blocks are needed to form one block of 1K-32 bits.

This same block, used by P1, will become a 4K-8bits. For P1, contrary to the usual organization, the two least significant address bits will be decoded to select one sub-block out of four.

In this organization, when P_1 is operating memory locations addressed consecutively are not physically consecutive, that is, two successive locations are not in the same sub-block. Looked at from a logical point of view, however, these locations appear consecutive (fig. 2). When P_2 is working all four consecutive locations of P_1 (from the logical point of view) appear as one location of P_2, and are access simultaneously by P_2. (fig. 3).

V - Generalisation of this method

The memory blocks can be distributed to several kinds of different processors (fig. 4). The organization (fig. 5) is facilitated if the following relationship holds :

(1) $C_1 \times W_1 = C_2 \times W_2 \ldots = C_n \times W_n = \text{constant}$

Wi is the word length of the processor Pi

Ci is the number of words in the block used by Pi

with : $W_i = W_{i-1}.2$ and $C_i = C_{i-1}.2^{-1}$

Consequently : (fig. 5)

(2) $C_n = 2.^{-1}C_{n-1} = 2.^{-2}C_{n-2} \ldots = 2.^{-n+1}C_1$

(3) $W_n = 2.W_{n-1} = 2.^{2}\, W_{n-2} \ldots = 2.^{n-1}W_1$

In the limit, $C_n = 1$ which determines the minimum size of the usable bloc for any processor P_i. In our case and from definition of C_n, in (2) above $C_n < C_{n-1} < \ldots . < C1$ we have :

$$W_n = 32$$
$$W_1 = 8$$

Then $\frac{Wn}{W1} = \frac{C1}{Cn} \Rightarrow C_1 = 4$

The minimum size for P_1 will be $C_1 = 4$

The formulas (1), (2) and (3) can be obtained from the binary address decoder. (fig. 6).

VI - How to synchronize the processor ?

We have many situations, all of which fall into two categories :

- Master - slave processors
- Symmetric processors.

The first configuration is not complicated to manage [8]. The master processor determines the distribution of the memory blocks for the benefit of the slave processor which, after execution, interrupts the master processor. Only status information is communicated on the input-output buses.

In the second solution, each processor keeps its autonomy and can use some of the other's resources but we must take care to avoid deallock [9].

Conclusion

Communication between processors in the method presented offers the advantage of speed and simple software. The hardware realization is simplified by using three-state gates to separate processor-buses. There is however a problem with the optimum size of a block, since minimum communication is equivalent to the size of a block, which is often imposed by the commercial RAMs. But the current low price of these allows us not to insist on the optimum size. Finally, this solution allows a very close logical interaction and from a physical point of view (real parallel processign and no bus sharing) an almost negligible interdependence.

AKNOWLEDGEMENTS

I wish to thank Professors J. VIGNES and C. GIRAULT, for their encouragement on this research and M. JAMMIER for helpful discussion about software.

References :

[1] Microprogrammed Multiprocessor Graphic Controller.
- A. BERNARDY - MICRO - 6 MARYLAND 1973.

[2] Design of a microprogrammed Alphanumeric terminal.
- F. DROMARD. MICRO. 7 Palo-Alto california 1974.

[3] Design of a Microprogrammed lexical Microprocessor - Y. CHU - MICRO 8 CHICAGO 1975

[4] A Firmware organization for error evaluation in numerical computations.
S.S.HYDER, V. UNG and J. VIGNES
Micro - 7 PALO - ALTO CALIFORNIA 1974

[5] Intel 8080 Microcomputer Systems

[6] MMI 4 Bit Expandable Bipolar Microcontroller

[7] A powerful Microprogram control-Unit - the 6700 Clive Ghest -
Micro 8 CHICAGO 1975

[8] A Microprogrammed Module for Musical Acquisition, Synthetical Replay and Edition.
C. APERGHIS, R. TERRAT, V. UNG.
Micro 8 CHICAGO 1975.

[9] A minimal connection between two symmetrical computers
M. CHEMINAUD, C. GIRAULT, V. UNG - Online international - conference LONDON 1975.

[10] Un système de communications : Logiciel ou Matériel ?
M. CHEMINAUD et A. SCRIZZI - Colloque Internationale sur la Programmation 1974 - PARIS.

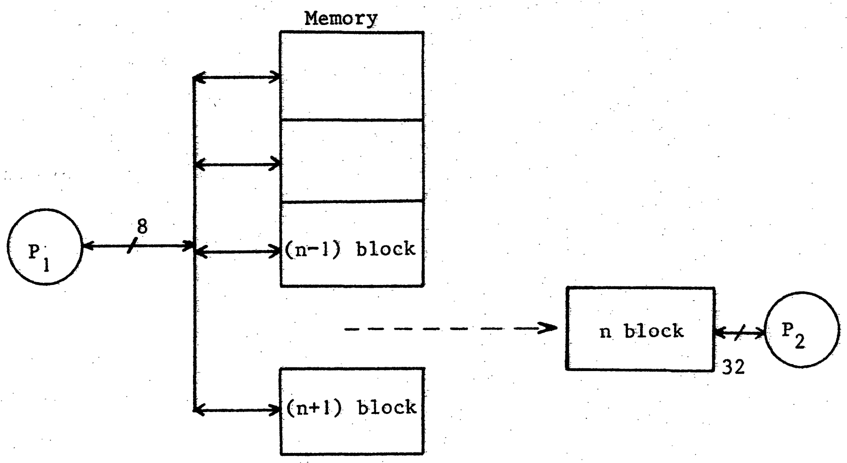


Fig 1 : The distribution of memory block n for use by processor P_2

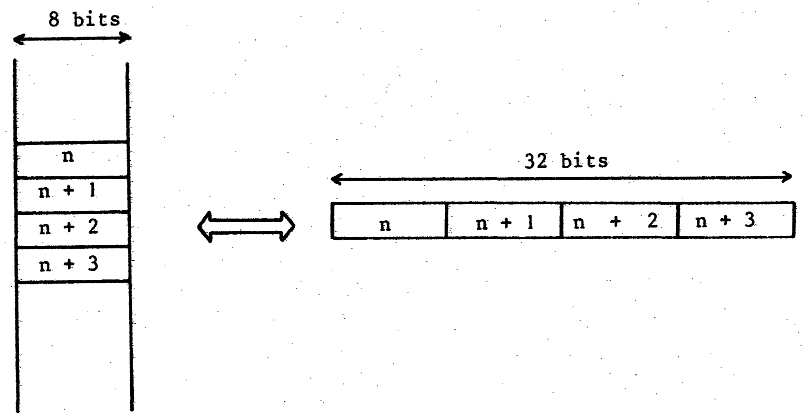


fig 2 : Logical representation

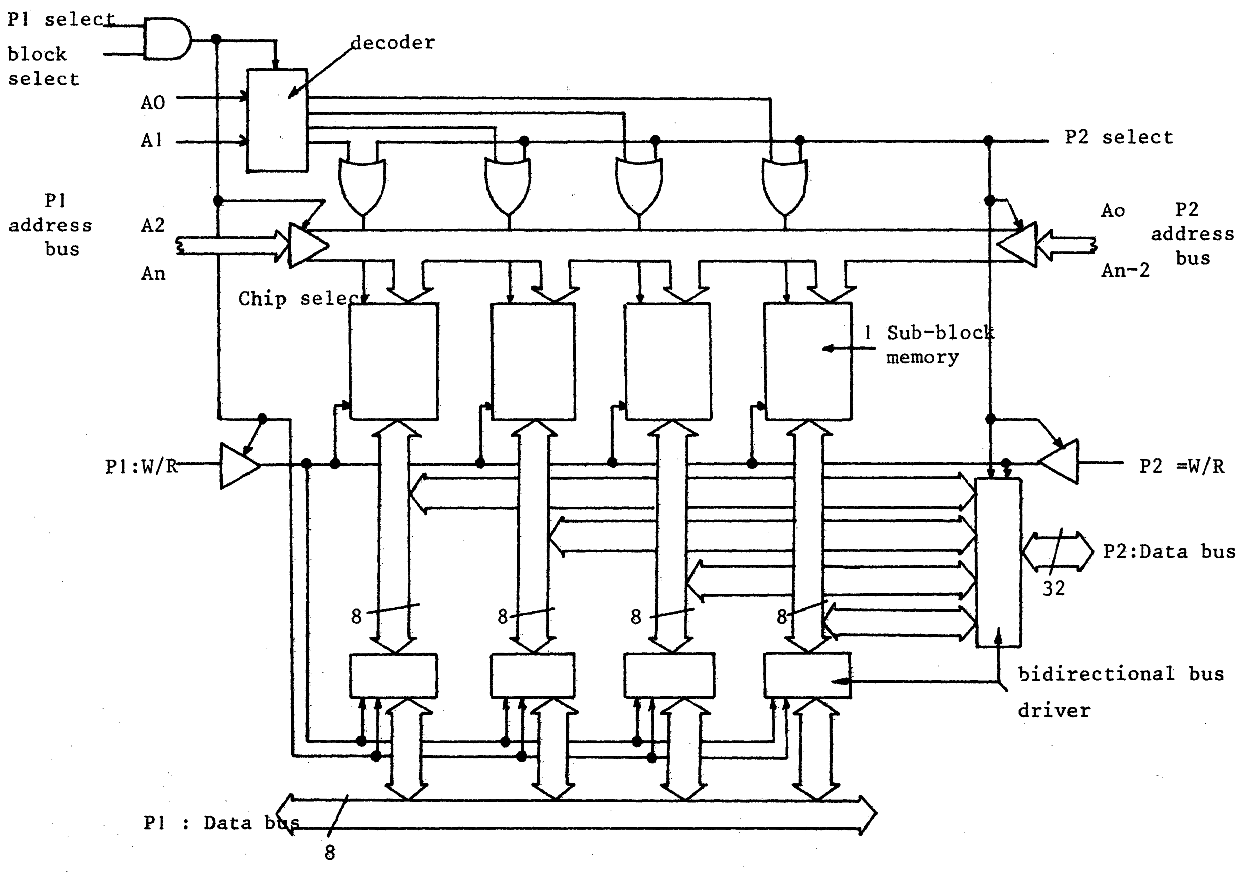


Fig 3 : Physical 8-32 bits organization

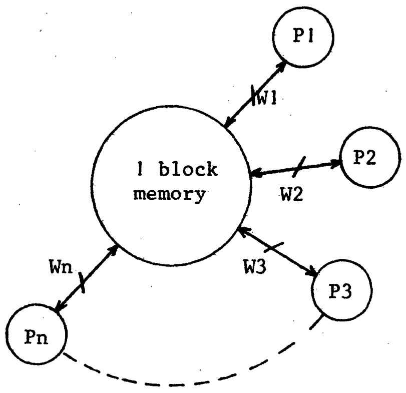


Fig 4 : A memory block used by any Pi with Wi

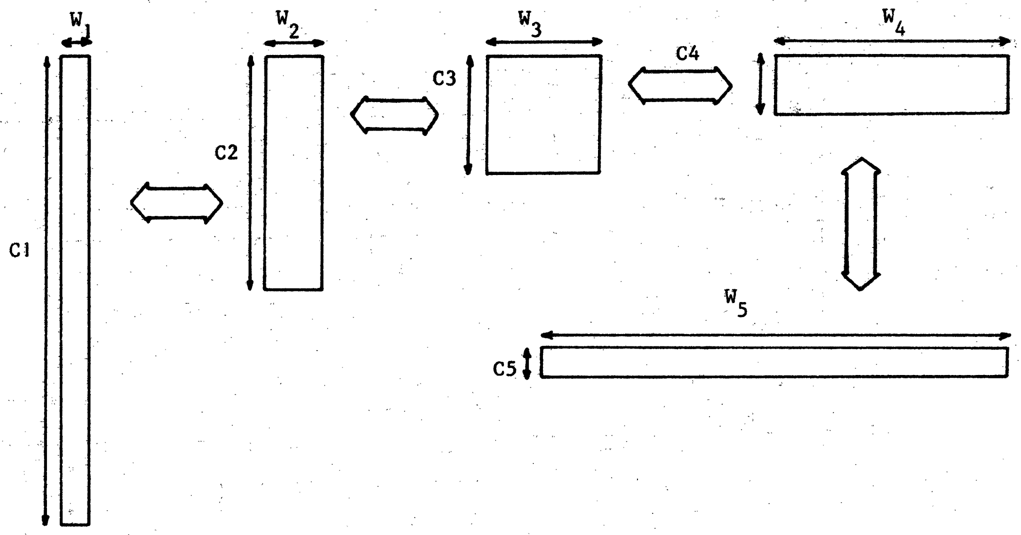


Fig 5 : Successive forms of a memory block

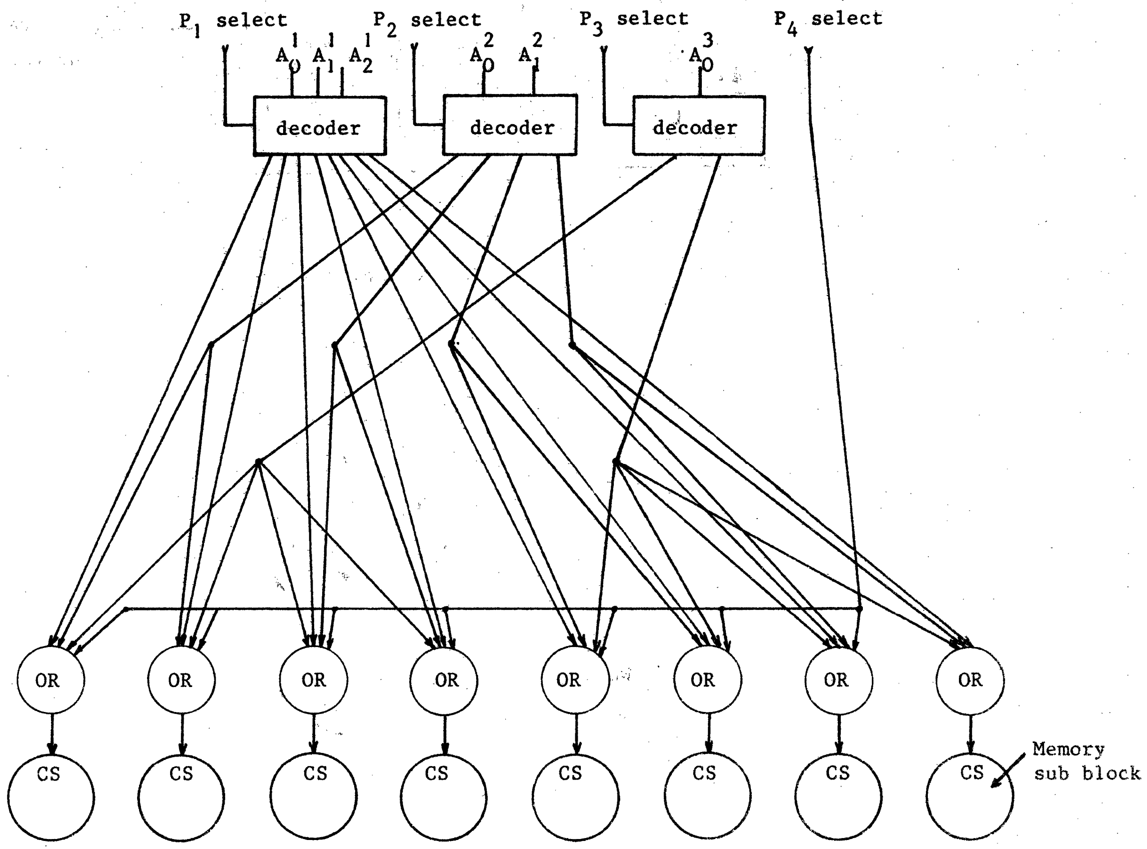


Fig 6 : Detail about chip select operation
($P1 \oplus P2 \oplus P3 \oplus P4 = 1$)

DESIGN CONSIDERATIONS IN MULTI-MINICOMPUTER PERFORMANCE

Tadaaki Bandoh
Yukio Kawamoto
Hitachi Research Lab. of Hitachi Ltd.
Hitachi-shi, Ibaraki-ken, Japan 319-12

Summary

This paper deals with the input output performance of multi-minicomputer structure. Fig.1 shows the typical structure of the multi-minicomputer. Each processor element (PE) has its own private memory (PM) and input output devices. The common memory (CM) and the common input output devices are shared by several processors. The common resource contention decreases the instruction execution rate [1] and also causes the over-run error.

The over-run error is caused by the waiting time exceeding the maximum allowance. Some input output devices, e.g. disks or drums (DMA) must transfer the data in proportion to their revolution. The waiting time is the time for which a DMA must wait until it can transfer the data.

In order to decrease the over-run error, it is necessary to make the waiting time shorter and the allowance longer. The influential designs are,

(1) Request selection logic in CM: This selects a PE of a DMA which accesses CM. The typical logic is priority or circular method.
(2) Acknowledge time: This is the time required for a PE to respond to the request of a DMA. In the multicomputer system, this is at least two memory cycles because of the test and set instruction which requires two memory cycles.
(3) Continuous transfer: This method allows a DMA to transfer data continuously if some datas are remained in the buffer.
(4) Buffer capacity: Larger buffer capacity of DMA allows longer waiting time. All DMAs should have the same allowance, or the DMA which has the larger allowance causes an over-run error of the DMA which has the smaller allowance.
(5) Interleaving of CM: This technique is effective to reduce the average waiting time, but not the maximum waiting time.

These factors decide the over-run probability, i.e. how many DMAs can run simultaneously.

Next, the quantitative data is examined in a simple case. Fig.2 shows the model of the analysis. There are 2 stages of the selection, i.e. the selection in a BUS and the selection in CM. Assume only PE accesses CM and DMA accesses only PM. DMA must wait while PE accesses CM. The access request of PE to CM is a closed loop and exponentially distributed. Rq is the probability that a PE requests during CM single cycle. The service time of CM is a unit time. The request selection logic in CM is either FIFO or priority or circular method. In this case, if the distribution of the waiting time is known, the over-run probability is obtained by using the time chart.

The waiting time distribution is calculated by using the concept of imbedded Markov Chain [2]. Let x(ti) be the state (i.e. queue size) when the i th caller service is completed. The calculation method is as follows.

(1) Make a transition matrix from x(ti) to x(ti+1)
(2) Calculate the probability of the entry position of a new request in x(ti)
(3) Follow the position of the new request in the transition matrix until it is serviced.

The result of the calculation is shown in Fig.3. The waiting time is normalized as the single service time is 1. The waiting time t(n-1<t≤n) is represented by time n. In the case of priority service, the waiting time of the lowest priority is shown. The result indicates that the circular service is not so different from FIFO and the priority service has disadvantage.

References

[1] Dileep P. Bhandarkar, "Analysis of Memory Interference in Multiprocessors", IEEE TRANSACTIONS ON COMPUTERS Vol. C-24, No. 9, Sep. 1975, pp 897-908

[2] Leonard Kleinrock, QUEUEING SYSTEMS, Vol.1 John Wiley & Sons, pp. 174

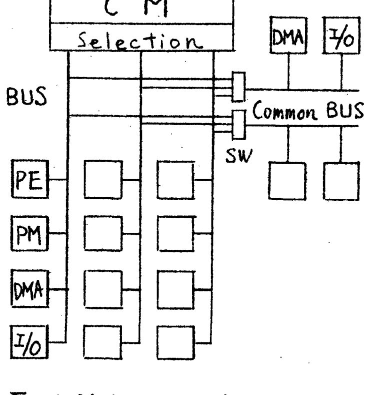


Fig 1 Multi-mini structure

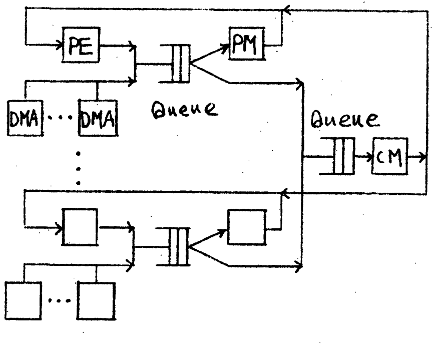


Fig 2 System model

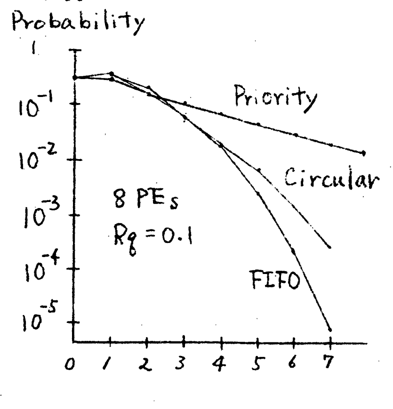


Fig 3 Waiting time distribution

A MODULAR VECTOR PROCESSING UNIT

S. R. Ahuja and J. R. Jump
Rice University
Houston, Texas 77001

SUMMARY

This paper presents and analyzes the architecture of a vector processing system. The system is best viewed as a functional unit similar to the pipelined arithmetic unit found in some current computers [1, 2, 3]. However, it differs in the following two significant ways from conventional pipelined units.

First, the proposed system is modular. It consists of several identical and independent modules. The external functional behavior does not depend on the number of modules and it will work properly with any number of modules. The only effect of changing the number of modules is to change the computation rate of the system. Thus modules can be added or removed without changing any applications or systems programs. This allows a straightforward tradeoff between hardware and performance. Moreover, the system exhibits fail-soft properties since if a faulty module is detected, it can simply be removed, resulting in a slight decrease in performance but no other changes.

Second, the system is programmable. Each of the modules is a general purpose processor (perhaps an LSI Microprocessor). Each operation performed by the system is defined by a microprogram. A copy of each such microprogram is stored in every module. Hence, the choice of operations is not limited to simple arithmetic operations. The operations can be chosen to fit the needs of the user and can be changed for different applications or when improved microprograms are developed.

The system consists of a number of processor modules which share two data busses; i) a common input data bus for the transfer of operands from an external memory to the processors, and ii) a common output data bus for the transfer of results from the processors to the memory. The control of each bus is distributed among the modules and essentially consists of a one bit shift register used to shift a single activation bit cyclically from one module to the next. Whichever module contains this activation bit has access to the bus. Thus the module containing the input activation bit will receive operands necessary to perform an operation. Once the operands are received, the operation is initiated and the activation bit is passed on to the next module. Similarily the module containing the output activation bit transfers the results of a previous operation, if any, to the memory, after which the activation bit is passed on to the next module. An activation bit is held at a processor only until the processor completes its data transfer.

This scheme allows i) the execution of several operations to proceed concurrently in different processors, and ii) the input and output phases (data transfers) of a processor module to be overlapped with the execution of operations in other processors. The input and output control loops work independently and asynchronously. Thus the output loop ensures that a processor module can output its results as soon as the module finishes an operation, irrespective of the execution time of that operation.

The performance analysis of the system provides a derivation of the throughout (i.e., the number of operations performed per unit of time) as a function of the number of modules, the number of operations to be performed, and the time required to perform a single operation in one module. Hence this analysis provides a quantitative measure of performance that can be used to determine the number of modules needed to achieve a given throughput. The performance of a system with N modules is shown to be the same or better than that of an equivalent N stage pipelined system [4]. In particular the total time of operation for a vector operation is given by

$$T = (Z+1)r + R + \left(\left\lceil \frac{Z}{N} \right\rceil - 1\right) \left(R \dot{-} (N-1)\, r\right)$$

where N = the number of processors, Z = the vector length, R = the processing time per processor, r = data transfer time (input and output), and $R \dot{-} (N-1)r = R-(N-1)r$ if $R > (N-1)r$ and 0 otherwise.

This tends asymtotically to $\left(\frac{R+r}{N}\right)Z$ for very large Z, showing an effective processing time of $\left(\frac{R+r}{N}\right)$ per operation. That is, the system exhibits a parallelism of order 'N' in that it processes the operations at a rate 'N' times that of a single processor.

REFERENCES

1 Anderson, S.F., et al, "The IBM System/360 Model 91; Floating Point Execution Unit," IBM J. Res. Develop., Vol. 11, pp. 33-53, January 1967.
2 "Control Data Star-100 Computer System Hardware Reference Manual," Technical Publication Dept., CDC, Arden Hills, Minn. 55512.
3 "ASC - A Description of the Advanced Scientific Computer System", Texas Instruments, Inc., April 1973.
4 Jump, J.R. and Ahuja, S.R., "Effective Pipelining of Digital Systems," Submitted for publication. February 1976.

A SHARED MEMORY TECHNIQUE FOR DIFFERENT MICROPROCESSORS

Dr. Ronald L. Krutz and Bob Reynouard
Department of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213

Summary

To investigate shared memory techniques two popular general purpose microprocessors were chosen as bases for investigation - Intel 8080 and Motorola 6800. The criteria for evaluating each technique was design simplicity, efficiency and programming limitations. The system, as imagined, was to provide each processor with local memory (ML), to which only that processor has access, and to interface the two processors through a block of shared memory (MS). The problem was to avoid simultaneous requests for shared memory by both processors.

Direct memory access was considered as a solution to the problem, but several shortcomings to this approach made it a second or third choice to the "wait state" technique described below. Of the two processors only the 8080 possessed the desired ability to be externally halted in the middle of an instruction. This feature served as the basis for the shared memory interface design. The 6800 is given top priority permitting it immediate access to MS. When both processors are vying for MS the 8080 is alerted of the 6800's request, then enters a "wait" cycle. The 6800 clock must be synchronized with the 8080 clock to insure sufficient time to alert the 8080 of a 6800 request for MS.

The technique proves efficient and easily/economically implement - an example for interfacing an 8080 and 6800 with 1K of shared memory costs under $50.

Good programming practice can achieve high 90 percentile efficiencies. Good programming practice means program mapping to allow each processor to operate in parallel as much as possible. An example, given in the main text, achieves 99% efficiency relative to each processor operating independently.

Parallel processing can prove to be an advantage in real time systems where execution time is critical. Another advantage of a shared memory system is shared I/O with common peripheries being accessed through MS address locations.

Three or more processors may also be configured to share memory using this "wait state" technique. The general requirements for multiprocessor shared memory communication are discussed further in the main text.

References

[1] Motorola Microprocessor Applications Manual, pp. 2-16

[2] Intel 8080 Microcomputer Systems User's Manual, (July 1975)

[3] Intel 8080 Assembly Language Programming Manual, p. 55

PARALLEL RECOGNITION OF PATTERNS: INSIGHTS FROM FORMAL LANGUAGE THEORY

Michael Moshell
The University of Tennessee
Knoxville, Tennessee 37916

Jerome Rothstein
The Ohio State University
Columbus, Ohio 43210

Abstract

We define bus automata, which are uniform arrays of finite automata ("cells") with modifiable channels through cells which allow long-distance communication. This permits separation of the functions of state-change (or switching) and information transmission, and analysis of their respective time costs. Most previous cellular automaton research does not make this distinction.

We define immediate languages as those formal languages accepted in a fixed number of steps by bus automata, regardless of the size of the input. Subfamilies within the immediate languages are described and compared to other parallel processing language hierarchies. From these comparisons we can infer some of the geometric and algebraic properties of language and pattern classes which admit rapid syntactic recognition by parallel cellular computing devices.

I. Introduction

One important model of parallel computation has been the cellular automaton (CA), introduced by Von Neuman (16) to study self-reproducing systems. We add to the CA a locally modifiable communication network, composed of binary channels through cells. The resulting bus automaton (BA) formalism allows us to design cellular computers with less concern for communications problems common to earlier CA research, such as the "firing squard synchronization problem" (17). The concept of the BA originated with Rothstein (12,13,18).

A bus automaton is a collection of finite sequential machines ("cells") arranged at the points with nonnegative integer coordinates in an n-dimensional Cartesian coordinate system (N^d). The inputs to a cell C are the outputs of neighboring cells in a standard neighborhood, the cells directly or diagonally adjacent to C. In d dimensions, C has 3^d-1 standard neighbors. All cells change state synchronously; the new state is a function of the old state and inputs.

Most previous CA research used Moore finite sequential machines (Moore fsm's) as cells. In formalizing the BA, we use Mealy fsm's with a particular type of output function (the "C-function") to represent binary channels through cells. This restriction also avoids the problems of indeterminacy for Mealy cellular automata pointed out by Hennie (2).

For brevity, complete proofs are omitted in this paper. Most of these proofs involve geometric arguments and many diagrams.

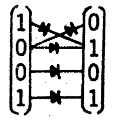
II. Bus Automata

C-Functions

A function $f:B_k \to B_k$ of Boolean (column) k-vectors is a C-function iff there exists a k by k Boolean matrix M_f such that, for b in B_k,

$f(b) = M_f.b$, where . represents Boolean matrix product. C-functions are precisely the functions realizable by passive diode networks. See the example in Figure 1.

For vector $b = (1\ 0\ 0\ 1)^T$,



$$= f(b) = \begin{pmatrix}0&1&0&0\\1&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}\cdot\begin{pmatrix}1\\0\\0\\1\end{pmatrix} = \begin{pmatrix}0\\1\\0\\1\end{pmatrix} = M_f.b$$

Figure 1: Example of C-Function ("conduction function")

C-functions easily generalize to m arguments, using m Boolean arrays.

For example, consider the binary C-function $H:B_3 \times B_3 \to B_3$ represented in Figure 2.

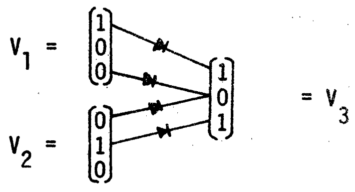


Figure 2: A Binary C-Function $H:B_3 \times B_3 \to B_3$.

H uses the connection matrices M_1, M_2 of m(=2) bipartite subgraphs. We have

$$H(v_1,v_2) = M_1v_1 + M_2v_2$$

$$\begin{pmatrix}1&0&0\\0&0&1\\0&0&0\end{pmatrix}\cdot\begin{pmatrix}1\\0\\0\end{pmatrix}+\begin{pmatrix}0&0&0\\1&0&0\\0&1&0\end{pmatrix}\cdot\begin{pmatrix}0\\1\\0\end{pmatrix}=\begin{pmatrix}1\\0\\1\end{pmatrix} = V_3$$

We define the set $C_{m,k}$ as the set of all m-ary C-functions from $(B_k)^m$ to B_k.

Bus Automata

We need the following definition. A Mealy finite sequential machine (fsm) is an ordered quintuple $M = (\Sigma, \Delta, Q, f, h)$ where

Σ is a finite set, the input alphabet;
Δ is a finite set, the output alphabet;
Q is a finite set called the state set;
f: $\Sigma \times Q \to Q$ is the state transition function
h: $\Sigma \times Q \to \Delta$ is the output function.

The standard neighborhood of a point Y is the set $\mathrm{NBHD}(Y) = \{W \varepsilon Z^d | W = (y_1+e_1, \ldots, y_d+e_d)$,

$Y = (y_1, \ldots, y_d), e_i \varepsilon \{-1,0,1\}$ for $1 \leq i \leq d$, and

$W \neq Y.\}$ Let $\mathrm{NBR}: N \times Z^d \to Z^d$ be any function such that for $1 \leq i \leq j \leq 3^d-1$, $\mathrm{NBR}(i,Y) \varepsilon \mathrm{NBHD}(Y)$, and if $i \neq j$, $\mathrm{NBR}(i,Y) \neq \mathrm{NBR}(j,Y)$.

A bus automaton (BA) is an ordered 7-tuple $M = (d,k,Q,f,g,G,q_0)$ where

1) d, a positive integer, is the dimension of the BA;
2) there is a defined a Mealy fsm

   $C = ((B_k)^{3^d-1}, B_k, Q, f, h)$ called a cell of M, with

   $(B_k)^{3^d-1}$ as input alphabet,
   B_k as output alphabet,
   Q as state set, (the "cell state set")
   f: $(B_k)^{3^d-1} \times Q \to Q$ the state transition function,

   h: $(B_k)^{3^d-1} \times Q \to B_k$ the output function;
3) g: $Q \to B_k$ is the local output function, and G: $Q \to C_{3^d,k}$ is the C-function selector function and h is defined in terms of g and G:

   for $V = (v_1, \ldots, v_{3^d-1})$ in $(B_k)^{3^d-1}$,

   $h(V,q) = G(q)(v_1, \ldots, v_{3^d-1}, g(q))$
4) $q_0 \varepsilon Q$ is the quiescent state, and

   $f((0,0,\ldots,0)^{3^d-1}, q_0) = q_0$ and

   $h((0,0,\ldots,0)^{3^d-1}, q_0) = (0,0,\ldots,0)$

Condition 3 (the definition of cell output function h) means that with each state q of a cell of M is associated a C-function G(q) which represents "channels" in cells. Channels transmit locally originated signals (g(q)), or signals from cell inputs $(v_1, \ldots, v_{3^d-1})$ to outputs without requiring state changes. Different C-functions may be associated with different states; the channels are thus modifiable by state changes.

Operation of Bus Automata

For a bus automaton $M = (d,k,Q,f,g,G,q_0)$ we define a state configuration of M as a function

$\gamma: N^d \to Q$ (i.e., an assignment of a state to every cell of M). γ is finite iff the set $\{Y \varepsilon N^d | \gamma(Y) \neq q_0\}$ is finite (i.e., only a finite number of M's cells are non-quiescent). An output configuration of M is a function $\omega: N^d \to B_k$. The sets $\Gamma = \{: N^d \to Q$ such that γ is finite$\}$ and $\Omega = \{\omega: N^d \to B_k\}$ are the sets of all finite state and output configurations of M, respectively.

When a given state configuration is established, the BA will then run through a series of output configurations, until all signals have reached their destinations ("settled"). The next state configuration may then be determined on the basis of these settled (or stable) values. We now formalize this.

For $Y \varepsilon N^d$, $\gamma \varepsilon \Gamma$, $\omega \varepsilon \Omega$, we write $<\gamma>_Y$ when we mean the state of cell C_Y, and $<\omega>_Y$ when we mean the output of cell C_Y. For $Y \varepsilon Z^d$ but

$Y \notin N^d$, we trivially extend all γ and ω; $<\gamma>_Y = q_0$ and $<\omega>_Y = (0,0,\ldots,0)$.

We define an output configuration history function $\Theta': N \times \Gamma \to \Omega$ as follows: for any

$\gamma \varepsilon \Gamma, <\Theta'(0,\gamma)>_Y = (0,0,\ldots,0)$
and for $j \geq 1$,

$<\Theta'(j,\gamma)>_Y = h(<\Theta'(j-1,\gamma)>_{\mathrm{NBR}(1,Y)}, \cdots,$

$<\Theta'(j-1,\gamma)>_{\mathrm{NBR}(3^d-1,Y)}, <\gamma>_Y)$

Informally, the value of $\Theta'(j,\gamma)$ is an output configuration of M which results from j applications of an "output updating operation" to a state configuration γ; γ is not changed during such a process. The "output updating operation" is just the application of output function h to each cell of M.

We now define the stable output function $\Theta: \Gamma \to \Omega$. for $\gamma \varepsilon \Gamma$, if there exists a positive integer t such that $\Theta'(t,\gamma) = \Theta'(t',\gamma)$, for all $t' > t$, let τ be the least such t. Define $\Theta(\gamma) = \Theta'(\tau,\gamma)$. If no τ exists, then $\Theta(\gamma)$ is undefined.

Informally, the value of $\Theta(\gamma)$ is the output configuration resulting from state confuguration γ after all "propagating signals" have reached their destinations, and all transients have settled. Hennie (2) showed that for most classes of cellular automata consisting of Mealy fsm's, it is undecidable if Θ is totally defined. Feedback loops may occur such that the outputs of some cells "oscillate":, and these loops are not always detectable. If, in state configuration γ, oscillation is occurring and is not somehow arrested, then $\Theta(\gamma)$ will be undefined, because no two successive output configurations are ever the same. We proved (10) that no bus automaton undergoes such oscillations during one clock interval.

We now define a state configuration update function $\Pi:\Gamma \to \Gamma$ as follows: for $\gamma\varepsilon\Gamma$, if $\Theta(\gamma)$ is defined then

$$<\Pi(\gamma)>_\gamma = f(<\Theta(\gamma)>_{NBR(1,\gamma)},\ldots, <\Theta(\gamma)>_{NBR(3^d-1,\gamma)}, <\gamma>_\gamma)$$

$\Pi(\gamma)$ is undefined otherwise.

We see that Π is just the "global analog" of f.

Let us define the state configuration history function $\Pi':N \times \Gamma \to \Gamma$ which for an initial state configuration γ specifies the configuration $\Pi'(m,\gamma)$ resulting after "m applictions of Π".

$\Pi'(0,\gamma) = \gamma$
$\Pi'(m,\gamma) = \Pi(\Pi'(m-1,\gamma))$ if the right side is defined
$\Pi'(m,\gamma)$ is undefined otherwise.

Physical Interpretations

The formalisms used to describe the operation of bus automata were motivated by physical considerations. It is intended that bus automata represent constructible electronic devices. The output configuration history function Θ' (recursively defined), represents the transmission of signals through the cells without cell state-change occurring. This means that signals are conducted through the channels represented by the C-functions, but the channels are not modified.

The stable output function Θ represents the signal values after they have been settled. The state transitions of cells, represented by the state configuration history function Π', only occur when Θ is defined. Thus, for BA for which $\Theta(\gamma)$ is defined for all finite state configurations γ, represents a constructable physical device.(a)

Let us denote by C_p ("cost of propagation") the time required for a signal to traverse a cell of some physical model of a bus automaton. That is, a change of input, where the cell-state does not change, results in a change of output after interval C_p.

Let C_s ("cost of state-change", or "cost of switching") denote the time required for a cell to change state, after a clock pulse, and to change the functions G_q and $g(q)$. C_s and C_p are both assumed to be greater than zero. During the state-change interval (of length C_s) the cell output is undefined. The state to which a cell changes is, of course, a function of its previous state and inputs at the instant of the clock pulse.

(a) Of course, systems where the non-quiescent part of space is very large may still not be practically constructable.

All cells are assumed to receive the clock pulse at the same time.(b)

Consider the task of recognizing (or computing some function of) a pattern, represented in a BA by cells' initial states, and having a maximal diameter ("span") of n cells. On the order of nC_p time is required, in general, before results can be obtained. The time is required to communicate the details of the extremities of the pattern.

Consider a bus automaton operating with clock period T. ($T \geq C_s$, by necessity). Let us say that clock pulses arrive at times tT for t = 1,2, ... In the time interval between $(tT + C_s)$ and $(t + 1)T$, signals propagate through cells at a time-cost of C_p per cell. If r is the range, or maximum distance a signal can travel before the next clock pulse, then

$$rC_p = (t + 1)T - (tT + C_s)$$
$$= T - C_s$$

or

$$T = C_s + rC_p$$

If a particular algorithm being studied requires access at each clock step to all the data in a region of span n, then range r must be at least n; so

$$T = C_s + rC_p \geq C_s + nC_p$$

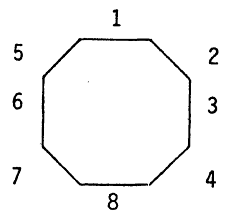
This constraint on physical models of bus automata is, of course, applicable to any physical realization of an abstract machine. (That is, signals must have time to reach their destinations.) The constraint requires re-emphasis here because bus automata (like other cellular automata) are extended in space so as to embody a parallel computation on an input of size n, where n may vary.

Geometric Notation for 1- and 2-dimensional Bus Automata

We represent the neighborhood of a particular cell by an octagon. Four connection faces are numbered 1 to 4 clockwise from the top; the remaining four are numbered 5 to 8 counter-clockwise. See Figure 3.

A neighbor cell of a cell B is referred to as the "i-th neighbor" of B if it touches the i-face of B. An incoming channel on face i is labelled Ri (R for "receiving") followed by a sequence number or symbol. Thus R6.1 is the first incoming channel on face 6. Outgoing channels are similarly labelled, using T ("transmitting"). The cell in figure 4 is in state q; its channels are represented in writing, as:

(b) We will not concern ourselves with problems of distributing the clock signal; since (unlike signals flowing through cells) its routing is uniform and unchanging, achieving simultaneous arrival is just a matter of delays.



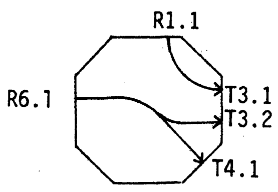


Figure 3. Cell Faces Figure 4. Channels

q: (R6.1; T3.2, T4.1)
(R1.1; T3.1)

Multiple channels with the same route may be grouped and referred to by a common name; they are drawn as double lines. See Figure 5.

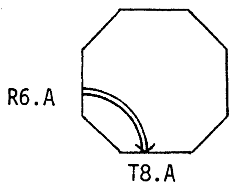


Figure 5. Multichannels

Equivalence of the graphic notation and the (two-dimensional) BA definition is shown in reference (10).

II. Immediate Languages

We intend for a BA to accept strings from some formal language in this fashion: the string is represented by the states of cells along one "edge" of the BA. The BA is run; if the origin cell ($C_{0,0,\ldots,0}$) ever enters a designated "acceptance state", then the string is accepted.

We consider a bus automaton $M = (d,k,Q,f,g,q_0)$. State set Q contains states E_1, and E_2 and designated "acceptance state" E_a. The cells $C_{1,0,\ldots,0},\ldots,C_{n,0,\ldots,0}$ in M are called the buffer of length n in M.

Consider any set $V \subset (Q-\{E_1,E_2,E_a,q_0\})$; we call V an input alphabet. For any string $X = x_1 \ldots x_n \varepsilon V^*$, the state configuration γ_X of M is defined as the function $\gamma_X : N^d \rightarrow Q$ such that:

$<\gamma_X>(0,0,\ldots,0) = E_1$ (E_1 is the left endmarker state;)

$<\gamma_X>(1,0,\ldots,0) = X_i$, $1 \leq i \leq n$ ($X = x_1 \ldots x_n$ is the input to M;)

$<\gamma_X>(n+1,0,\ldots,0) = E_2$ (E_2 is the right endmarker state;)

$<\gamma_X>Y = q_0$ for all other $Y \varepsilon N^d$.

We say that M accepts X iff there exists a positive integer r such that $<\Pi(r,\gamma_X)>_{(0,0,\ldots,0)} = E_a$. The language accepted by M, L(M), is the set of all strings $X \varepsilon V^*$ accepted by M.

For a bus automaton M and its state configuration history function Π', L(M) is immediate on M iff there exists a positive constant K_L such that, for all $X \varepsilon L(M)$, $<\Pi'(m,\gamma_X)>_{(0,0,\ldots,0)} = E_a$ for some $m < K_L$.
For any formal language L, if there exists a bus automaton M such that L is immediate on M, then L is an immediate language ($L \varepsilon IML$). Thus an immediate language is a language L accepted by some BA in at most K_L steps, regardless of the length of the input string.

Consider a bus automaton, M; and its stable output function $\Theta : \Gamma \rightarrow \Omega$. Using Θ we define a function $P : \Gamma \rightarrow N$ as follows: for $\gamma \varepsilon \Gamma$, $P(\gamma) = \tau =$ the least t such that $\Theta(\gamma) = \Theta'(t,\gamma)$ if $\Theta(\gamma)$ is defined, and $P(\gamma)$ is undefined otherwise. The value of $P(\gamma)$ represents the number of iterations of the output configuration update function required for the outputs to stabilize after M enters state configuration γ. We call $P(\gamma)$ the propagagation time for γ.

For M a bus automaton, if L = L(M) is immediate on M then L is linear propagation time immediate ("L-immediate") on M iff there exists a positive constant k such that for all strings X of length n in L,

$$\underset{0 \leq m \leq K_L}{\mathrm{MAX}} P(\Theta(\Pi'(m,\gamma_X))) < kn.$$

For any formal language L, if there exists a bus automaton M such that L is linear propgation time immediate on M, then L is a linear propagation time immediate language ($L \varepsilon LIML$). We sometimes say that M accepts L in L-immediate time.

For M a bus automaton, if L = L(M) is immediate on M then L is polynomial propagation time immediate on M iff there exists a polynomial function $f : N \rightarrow N$ such that for all strings X of length n in L,

$$\underset{0 \leq m \leq K_L}{\mathrm{MAX}} P(\Theta(\Pi'(m,\gamma_X))) < f(n)$$

For any formal language L, if there exists a bus automaton M such that L is polynomial propagation time immediate on M, then L is a polynomial propagation time immediate language ($L \varepsilon PIML$).

The set of languages immediate on bus automata of d dimensions will be called IML_d.

Using the previous definitions with obvious extensions, we can define the following families of languages:

IML; IML_d for all $d > 0$

$PIML$; $PIML_d$ for all $d > 0$

$LIML$; $LIML_d$ for all $d > 0$

The following inclusions follow directly from the definitions.

$$\begin{array}{ccc} IML_{d+1} & \supset & IML_d \\ \cup & & \cup \\ PIML_{d+1} & \supset & PIML_d \\ \cup & & \cup \\ LIML_{d+1} & \supset & LIML_d \end{array}$$

Examples of Immediate Languages

Theorem: the family of languages which are immediate on one-dimensional bus automata is exactly the family of regular languages. In fact, $IML_1 = PIML_1 = LIML_1 = REG$, the class of regular languages.

The technique used in the proof of the inclusion of REG in IML_1 will be briefly sketched here. The proof that IML_1 is included in REG is based on a theorem by Hennie (3) concerning linear-time languages and Turing machines. Both complete proofs are found in reference (9).

Consider a finite-state automaton, M, with the state transition diagram shown in Figure 6. The double circle represents the accepting state, and the feathered arrow designates the start state.

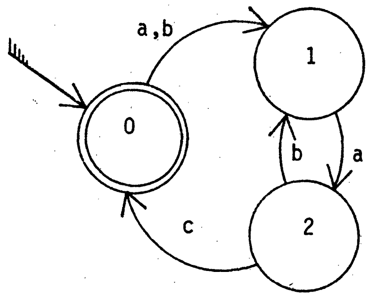


Figure 6. Finite Automaton M.

M accepts the regular language L = $((a+b)(ab)^*ac)^*$ over the alphabet V = {a,b,c}. We now design a bus automaton to accept L. To each letter x of V we assign a state q_x, with channels as shown in Figure 7. The "left inputs" of a cell correspond to the state of M.

Thus, if a letter a takes M from state 0 to state 1, a cell in state q_a contains a channel connecting input 0 to output 1. An input string $X = x_1...x_n$ is placed in the BA by setting cells $C_1,...,C_n$ to states $q_{x_1},...,q_{x_n}$. When the BA is activated, cell C_0 originates a signal (symbolized by * and ... in Figure 8) which flows through the channels, tracing a "state history" of M with string X as input. At the right end, cell C_{n+1} is in special state E_2. If the input to this cell corresponding to the acceptance state of M (in this case, input 0) receives a signal, the signal is sent back to C_0. Then C_0 enters state E_a and string X is accepted. This process requires only one state change by the BA, regardless of the length of X.

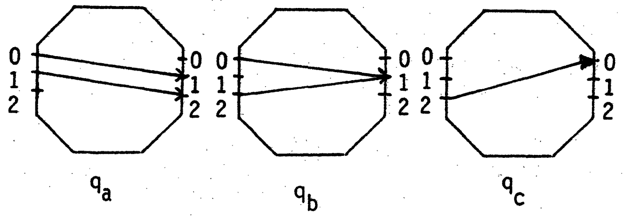


Figure 7. States of Corresponding Bus Automaton

The proof consists of a general treatment of this idea, which resembles the Krohn-Rhodes (6) semigroup representation of a finite automaton.

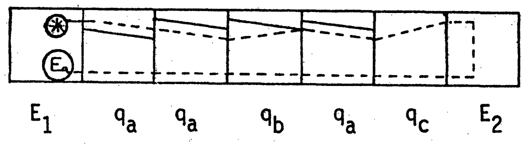


Figure 8. Operation of Bus Automaton Accepting L

Linear Languages

The linear context-free languages are those for which there exists a grammar $G = (V_n,V_t,P,S)$ whose productions are all of the form

$A \to aBb$ where $A,B \in V_n$,

or $a,b \in V_t \cup \{\lambda$, the null symbol$\}$

$A \to c$ $c \in V_t$

Languages such as $\{a^n b^n | n \geq 1\}$ are linear languages. The linear languages are the "two-sided" analogues of the regular languages. Since a one-dimensional BA accepts any regular language, we might expect that given any linear language L, there exists a two-dimensional BA to accept L. This turns out to be the case.
Theorem: the linear languages are in $LIML_2$.

Proof: Reference (9).

Dyck Languages

Consider three alphabets $A_m = \{a_1,...,a_m\}$,

$A'_m = \{a'_1,\ldots,a'_m\}$, and their union P_m. The Dyck language D_m is the set of strings in P_m^* that can be reduced to the empty string by successive deletion of substrings $a_i a'_i$, $1 \le i \le m$. Informally, language D_m is isomorphic to the set of "correctly balanced strings" of m kinds of parentheses, e.g. $D_2 \cong \{(),<>,()<>,(<>),\ldots\}$.

Theorem: For any positive integer m, a two-dimensional bus automaton M_m can be constructed which accepts Dyck language D_m in L-immediate time.

An example of the operation of M_m is given. The proof is found in reference (9).

Let m = 2. For readability replace a_1, a'_1, a_2, a'_2 by (,),<,>, respectively; and call the language D'_2. The string X = $(<()><>) \in D'_2$ is placed in the input buffer of M_2. When M_2 is started, signals are sent upward on coordinate busses y_1 and y_2 (see Figure 9) by cells of the buffer containing "left parentheses" and "right parentheses", respectively.

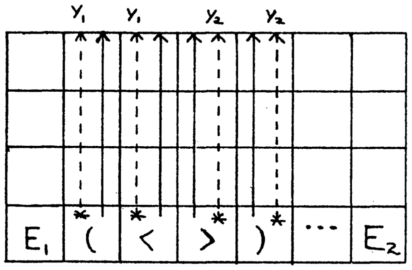


Figure 9. y Signals

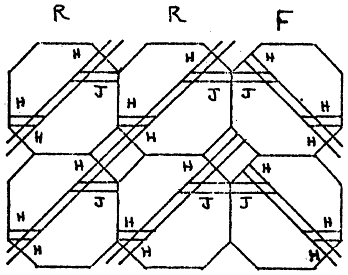


Figure 10. R and F States

Quiescent cells receiving signals on busses y_1 and y_2 enter states R ("rising") and F ("falling") respectively, with multichannels H and J, as in Figure 10.

An input buffer cell containing a "left parentheses" sends out a signal representing that symbol, via the bus formed of H and J multichannels.

If X is well-formed, each cell storing a "right parentheses" receives a signal, via the H-J bus, representing the corresponding left symbol. The cell then enters state Z, Figure 11.

Cells in state Z send a signal on a single-channel bus H'-J', which connects the same set of cells as are connected by the H-J bus; but signal propagation is in the opposite direction. For brevity, we say that H'-J' is antiparallel to H-J.

The cell originating the left symbol signal thus receives a return signal, which indicates correct pairing of symbols. This cell also enters state Z. See Figure 12.

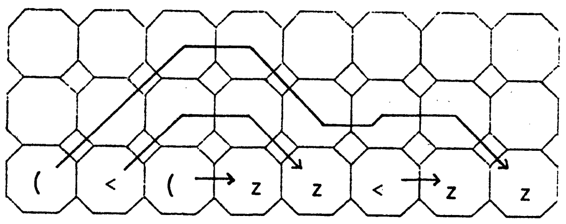


Figure 11. Right Paired Cells Enter State Z.

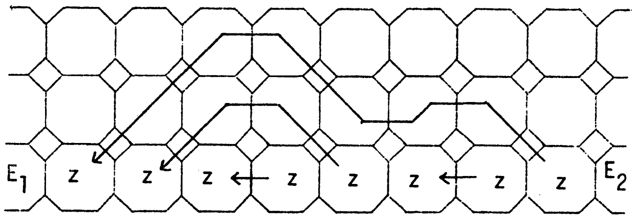


Figure 12. Left Paired Cells Enter State Z.

If the entire input buffer enters state Z then an "acceptance signal" flows from the cell in state E_2 to the origin cell, in state E_1. The latter then enters state E_a, and M_m accepts X.

Other Immediate Languages

We have shown that:

IML_1 is precisely the class of regular languages

$LIML_2$ includes the linear context-free languages, the Dyck languages, and many other context-free languages;

$LIML_3$ includes $L_p = \{a^p | p \text{ is prime}\}$

$L_s = \{a^s | s \text{ is square}\}$

and many other non-context-free languages.

Closure Properties of Immediate Languages

All the IML families (IML,PIML,LIML, and IML_i, $PIML_i$, and $LIML_i$ for $i \ge 1$) are closed under Boolean operations, reversal, intersection with regular sets and inverse homomorphism; none (except IML_1 = REG) are closed under homomorphism. The families IML, PIML and LIML are closed under Kleene (star) closure. Proofs of all these are found in reference (9).

III. Relationship to Work of Smith

Smith (15) studied a type of one-dimensional Moore CA, called bounded cellular spaces (BCS). Language families RDBCS and LDBCS are the languages accepted in real-time [c] and linear-time [c], respectively by deterministic BCS. RDBCS includes all examples so far found in LIML3. We proved (9) inverse homomorphic closure for LDBCS; the closure properties of $LIML_3$ and LDBCS are then identical. It is tempting to speculate that $LIML_3$ and RDBCS (or LDBCS, perhaps) are the same set of languages where $LIML_3$ represents a "factoring out" of the communication problem from the computation problem, arranging the communication in extra spatial dimensions rather than using cell-state transmission. The chief obstacle to this conjecture is that the "λ-move-free deterministic CFL's" [d] are in RDBCS but have not been found in $LIML_3$. This class is defined by acceptance on a push-down automaton, a sequential operation more easily simulated on a BCS than a BA.

IV. Context-Free Languages (CFL)

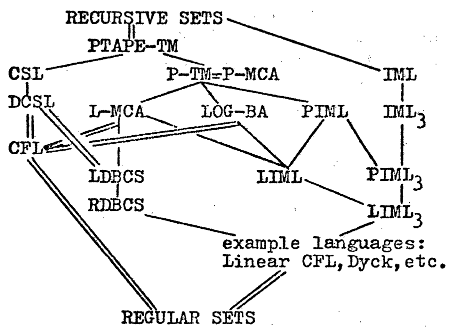
Cole (1) studied "iterative arrays of finite automata" (IFAs), which are Moore cellular automata with a sequential input (one symbol per clock interval) to some single cell, e.g. the origin. He showed that some non-CFL's could be recognized by an IFA in real-time, and that there exist CFLs which no IFA can recognize in real-time. Kosaraju (5) showed that any CFL can be accepted by some 2-dimensional IFA in $(1+\varepsilon)$ real time for any $\varepsilon > 0$.

We have not determined the relationship between CFLs and the IML families. However we have shown (9) that for any CFL, a BA can be constructed which accepts its strings in a number of clock intervals proportional to the logarithm of string length. It is also not known if a 1-dimensional CA (of any type) can be built to accept a given CFL in linear time, but most evidence suggests not.

Relationship of the IML families to other language families is shown in the following diagram. Sets are included in sets above them to which they are connected. Double lines indicate that proper inclusion has been established.

(c) I.e., in a number of clock intervals equal to (real-time) or proportional to (linear-time) the length of the input string.

(d) Languages accepted by a deterministic PDA which moves its input head with every controller state-change.



CSL = context-sensitive languages
DBCS = deterministic bounded cellular space
DCSL = deterministic CSL
IML = immediate language
L- = linear-time
LOG-BA = logarithmic number of state changes in a BA
MCA = Moore cellular automaton
P- = polynomial time
PTAPE = polynomial tape
RE = recursively enumerable
TM = Turing machine

Figure 13. Inclusion Among Language Families.

V. Conclusions

Observation 1:

Neither the DBCS nor the IML families correspond well to the "Chomsky hierarchy" of formal languages. The context-free languages have not been included in LDBCS, nor in an IML family; neither do they correspond to any naturally occurring class of IFAs. However, many IML languages (even in $LIML_2$) are not context-free. Parallel recognition-defined languages simply represent a different partitioning of the formal languages than do the sequentially defined "traditional" formal languages. For this reason also, we can predict that array grammars (8) will not generate classes of patterns which are easily recognized in parallel, since their "degenerate" (one-dimensional) cases must correspond to the Chomsky-hierarchy languages.

Observation 2:

Considering a variety of "parallel" formalisms for generation of languages, we see that none of them correspond to an IML or DBCS family. None of the L-system languages in (4) are closed under inverse homomorphism; all the IML families are so closed. The parallel CFL's (14) and the absolutely parallel languages (11) are closed under arbitrary homomorphism, whereas no IML or DBCS family is so closed. It is likely that few if any deterministic parallel-acceptance classes will be found which correspond to parallel-generated classes of patterns. Parallel generation formalisms can create diversity in patterns so rapidly that it seems clear that, even using parallel devices, only a non-deterministic recognizer could recognize the patterns in time commensurate with the time used to generate them. Proofs in this area are tantamount to solving the P = NP problem. Results such as Observation 2 reinforce the conviction that parallel generation systems produce pattern classes which are yet another partitioning of the set of patterns, not simply related to parallel or sequential recognition.

Observation 3:

Languages so far found in IML and DBCS frequently have geometric or arithmetic descriptions; for instance L_p and L_2. (But note that the arithmetic is in "base 1" notation. The numbers involved are the lengths of the strings, not the strings interpreted in a radix system. Thus they are one level of "interpretation" closer to geometry than are, say, expressions of analytical geometry.) The speed with which these languages are recognized is partially a result of the facility cellular computers have in "spreading a computation out in space".

At a deeper level, these successes result from the existence of descriptions of geometry and arithmetic in terms of associative, distributive (sometimes commutative) operations. Relationships can be seen between this work and that of Kuck and Muraoka (7) and others, on parallel arithmetic. There also, these "tractibilities" (associativity, etc.) allow arithmetic expressions to be reorganized into equivalent expressions whose tree representations are broader and less deep. In our work, addition's associativity allows, for instance, multiplication to be represented by a collection of sets of additions simultaneously performed on variables represented by a signal's position in cellular space. These methods extend to some non-numeric computations such as Dyck and linear language recognition. It is unclear how many other "non-geometric" computations will be facilitated by cellular computers.

Acknowledgements

Portions of this research were performed under the support of a University Fellowship at Ohio State University.

References

1. Cole, S. N., "Real-time computation by n-dimensional iterative arrays of finite-state machines". IEEE Trans. Computers. C-18, Nr. 4, April 1969. 349-365.

2. Hennie, F. C., Iterative Arrays of Logical Circuits. MIT Press, 1961.

3. _____, "One-tape, off-line Turing machine computations". Info. & Ctrl. 8, 553-578 (1965).

4. Herman, G. T., "Closure Properties of some families of languages associated with biological systems". Info & Ctrl 24, 101-121 (1974).

5. Kosaraju, S. R., "Speed of recognition of context-free languages by array automata". SIAM J. Computing 4, 331-340, September 1975.

6. Krohn, K. and Rhodes, J., "Algebraic theory of machines. I. Prime decomposition theorem" Trans. Amer. Math. Soc. 116, 450-494, 1965.

7. Kuck, D. J. and Muraoka, Y., "Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity". Acta Informatica 3, 203-216 (1974).

8. Milgram, D. L. and Rosenfield, A., "Array automata and array grammars". Proc. IFIP 1971 Congress, Yugoslavia.

9. Moshell, J. M., Parallel recognition of formal languages by cellular automata. Ph.D. Dissertation, Ohio State University, 1975.

10. _____ and Rothstein, J., Bus Automata, Tech. Rpt. CS-76-14, Computer Science Department, University of Tennessee, March 1976.

11. Rajlich, V., "Absolutely parallel grammars and two-way finite-state transducers". J. Cptr. Sys. Sci. 6, 324-342 (1972).

12. Rothstein, J., "Patterns and Algorithms". Ninth IEEE Symp. on Adaptive Processes: Decision and Control". Austin, Texas, December 7-9, 1970.

13. _______. "On the Ultimate Limits of Parallel Processing." International Conf. on Parallel Processing, Aug. 23-26, 1976. Detroit, Mich.

14. Siromoney, R. and Krithivasan, K., "Parallel Context-Free Languages". Info. & Ctrol. 24, 155-162 (1974).

15. Smith, A. R., "Real-time language recognition by one-dimensional cellular automata". J. Cptr. Sys. Sci. 6, 233-253, 1972.

16. Von Neumann, J. (A. W. Burks, Ed.), Theory of Self-Reproducing Automata. University of Illinois Press, 1966.

17. Waksman, A., "An optimum solution to the firing-squad problem". Info. & Ctrl. 9, 67-78, 1966.

18. Weiman, Carl and Rothstein, J., "Polyautomaton Design for Recognizing Certain L-System Languages by Parallel Computation". 1975 Sagamore Cptr. Conf. on Parallel Proc., August 19-22, 1975, Rquette Lake, New York.

SOME COMPUTATIONAL AND SYSTEM THEORETIC PROPERTIES OF REGULAR PROCESSOR NETWORKS

Renato M. Ermann and William I. Grosky
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract -- The interconnection of a large number of microcomputers is becoming an attractive option. However, the knowledge of how to effectively utilize such networks is still rudimentary. In this paper we first demonstrate that regular processor networks can be used to solve within polynomial-time, some problems for which conventional computers have most probably no better than exponential-time algorithms. A general formulation in the propositional calculus that permits examination of the dynamical behavior of those networks is then introduced. The questions of completeness, controllability and reproducibility, with the optional stipulation that they preserve a given set of properties, can be analyzed by using that tool. The concept of a periodic state configuration is introduced and the evolution of periodicity throughout time in a uniform machine is characterized. Some necessary conditions on the neighborhood structure and set of hardware instructions of a uniform machine that executes a given algorithm are finally determined.

Introduction

In this paper, we explore some characteristics of homogeneously interconnected networks composed of identical processing elements. The definition of a regular processor network is introduced as a simplified model for this class of parallel systems. A group-graph formulation is chosen to represent the interconnection structure. It allows us to analyze apparently very different kinds of networks under a unified framework. Topologies such as rings, trees and square, triangular, hexagonal cellular structures are represented by groups. Most of the fundamental results concerning regular networks are due to Yamada and Amoroso [1] - [3] and Smith [4] - [5].

A regular processor network (RPN) is a 4-tuple (A, P, NI, I) where A is a finite, nonempty set called the state alphabet; P is a set of points called the cellular space, that associates with a binary operation ·, so that together they define a group; NI is an ordered set of n points referred to as the neighborhood index, and I is a nonempty set of functions called the admissible local transformations. The set A consists of the states which may be assumed by each individual identical processor (called a cell) in the cellular space; NI describes the regular interconnection pattern by specifying for each cell in the space, the set of cells directly connected to it (its neighbors); set I may be thought of as the collection of hardware instructions built into the network.

The group-graph formulation defines structures such that the neighborhood patterns look the same when viewed from any cell. A group is a set P and a binary operation · on P, such that associativity holds, there is a unique identity element e and every element of P has a unique inverse. Directed graph D = (G,A) with point set G and arc set A is called the group-graph of group G if it satisfies: $p_i, p_j \in G$, $(p_j, p_i) \in A$ iff $p_j p_i^{-1} \in H$, where H is the set of generators of group G. If H is a set of elements of group G and if all elements of G can be expressed as compositions involving only elements of H and their inverses, then H is the set of generators of group G. Figures 1 and 2 illustrate a square and an hexagonal cellular group, respectively.

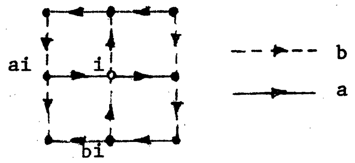


Fig. 1. Square cellular structure.

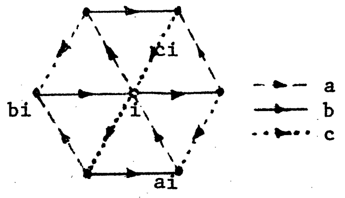


Fig. 2. Hexagonal cellular structure.

The defining relation for the group in Fig. 1 is $(ab)^2 = (a^{-1}b)^2 = e$, where a and b are the generators. Relation abc = cba = e specifies the group in Fig. 2. If the cellular space is Cartesian, the RPN is called a d-dimensional tessellation processor network (TPN). Let the neighborhood index NI be NI = $(w_1, w_2, .., w_n)$. Then N(i,NI), the neighborhood of cell i with respect to NI is given by N(i,NI) = $(w_1 i, w_2 i, .., w_n i)$. A contiguous, scope-n neighborhood in Z^1 is defined as NI = (k,k+1,..,k+n-1), where Z is the set of integers and k is an integer. It is convenient to include in the state alphabet two specially designated states: the quiescent state, denoted by 0, and the boundary state, denoted by B. A cell in state B remains permanently in state B and no cells in boundary states can be created after time zero. If all the neighbors of a quiescent cell are quiescent, then it will remain in state 0. The set of cells not in state B constitute the active cells. A space state configuration is an arbitrary mapping from P into A. If i is a cell in P, and the space is in configuration c, then c(i) is the current state of the processor located at cell i. By the state of the neighborhood of cell i in configuration c we mean the ordered set c(N(i,NI)). A configuration is called finite if and only if c(i) = 0 for all but finitely many cells i.

The operation of a RPN is specified by local transformations which produce the next state of each cell in P in terms of the state of its neighborhood. The simultaneous invocation of the same local transformation to the state of the neighborhood of every cell in the cellular space defines a global transformation of the current configuration into the next. This mode of operation will be called uniform and the RPN will be said to have a single-instruction-multiple-data-stream. When different local transformations are applied to different cells, the type of processing will be called non-uniform and the RPN will be said to have a multiple-instruction-multiple data-stream.

The fourth component in the quadruple is now defined. The set of admissible transformations I is any nonempty subset of the set of all local transformations definable from A, P and NI. If I contains a single transformation, the RPN is said to be monogenic. If it contains at least two, it is said to be polygenic. In the latter case a sequence of transformations is required to specify a particular computation of the RPN. We say that two configurations c_1 and c_2 are shift-equivalent if and only if there exists a u such that for any cell i in P, $c_1(u \cdot i) = c_2(i)$. The equivalence classes determined by the relation of shift-equivalence are called patterns.

We say that a RPN is controllable, if it is possible to transfer it from an initial configuration to any of a set of final configurations in some finite lapse of time through a sequence of admissible transformations. Reproducibility is the ability of a RPN to generate any member of a given class of configuration sequences. If a RPN can be transferred from a certain canonical starting configuration to any finite configuration in the space, with every transformation being admissible, it is said to be complete.

Let a property Q be a subset of the set of all configurations. A global transformation preserves property Q if a configuration has property Q if and only if its successor has property Q.

Computational Power

The Classes of Problems NP and P

Smith [4] and Seiferas [6] have presented some cases of computations for which TPN are faster than Turing machines, i.e., conventional computers. We further establish the computational potential of RPN by demonstrating that they can be used to compute within polynomial-time, answers for which there is broad evidence that Turing machines have no better than exponential-time algorithms.

More precisely, let us define NP (respectively P) to be the class of problems solved within polynomial-time by nondeterministic (respectively deterministic) multitape, multihead Turing machines. Many important problems which are not known to be in P are in NP. Karp [7] has provided strong evidence that the two classes may not be the same, by showing that many problems in NP would be in P if and only if P and NP were identical. The equivalence class of problems in NP having this property is called polynomial-complete. Either all of them admit some polynomial-time Turing algorithm or none of them does, and none is currently known. Polynomial-complete problems include testing the satisfiability of a propositional calculus formula in conjunctive normal form, traveling salesman, determining the maximum clique or minimal coloring of a graph, scheduling, register allocation, integer programming.

An algorithm for a problem in NP can be regarded as a procedure which, when confronted with a choice between (say) two alternatives, can create two copies of itself, and follow up the consequences of both courses of action. There is some constant k such that there are no more than k choices of next move in any situation. Repeated splitting may lead to an exponentially growing number of copies. Each sequence of moves leading to a halt of the nondeterministic Turing machine that executes the algorithm is of polynomial length. Thus, a problem in NP can be computed by a deterministic Turing machine through a backtracking search of polynomial bounded depth that takes exponential-time.

Formally, a sequence of up to t(n) moves of the non-deterministic machine M_1, where n is the size of the input, is represented by a string over the alphabet $\Sigma = \{0,1,\ldots,k-1\}$ of length up to t(n). A deterministic machine M_2 simulates M_1 on an input x of size n as follows. M_2 successively generates all strings v in Σ^* of length at most t(n) in lexicographic order. There are no more than $(k+1)^{t(n)}$ such strings. As soon as a new string is generated, M_2 simulates σ_v, the sequence of moves of M_1 represented by v. If σ_v causes M_1 to halt (generating a solution), then M_2 also halts. If σ_v does not represent a valid sequence of moves by M_1 or if σ_v does not cause M_1 to halt, then M_2 repeats the process with the next string in Σ^*.

A One-Dimensional TPN

We now demonstrate that any problem in NP can be solved within a time proportional to a polynomial of the input size, by using a deterministic exponential-space 1-dimensional TPN with a special outputting cell that has every active cell as its neighbor. By applying an effective bounding schema and performing interprocessor communication, an exponential number of active cells will only be needed in an insignificant number of cases. Initially, each of a possibly exponential number of active cells simulates a sequence of moves σ_v of the nondeterministic Turing machine, as described before. Within polynomial time, each of the cells will have either found a tentative solution or failed, and if there is a bounding schema, rated it according to a merit function. For example, in the case of the minimal coloring problem a tentative solution would be a coloring that is not necessarily minimal. The merit function would reflect the chances a certain coloring (possibly partial) has of being optimal, by determining a lower bound on the cost of a tentative solution. When a bounding schema is included to reduce the search process, the current bound propagates throughout the cellular space, thus discarding tentative solutions which do not meet it and diminishing the required number of active cells. The selection of a solution demands that each cell be able to compare its tentative solution with the rest. The outputting cell would display (by convention) the state of the leftmost cell which has a tentative solution of highest merit. This may be done either in unit time by means of a combinational circuit, or in polynomial time by performing a process similar to a binary search.

The essential feature of processor networks as compared to multihead, multitape Turing machines that permits the improvement in throughput is, in this case, the possibility for space-unbounded propagation of information. The most appropriate technique for implementing the outputting schema seems to be a packet switching broadcast communication mechanism.

Although there is the theoretical asymptotic limitation in propagation speed given by the speed of light, it is quite clear that in practical cases that bound will not be approached. As a matter of fact, since in general the time needed for propagation of information between two cells is negligible compared to the actual computation time, if we eliminated the outputting cell an exponential growth in total processing time would only start occurring for very large inputs.

Tree-Structured RPN

When tree-structured regular processor networks are used, the need for a distinguished cell to perform the input/output process disappears. A regular tree of fanout k can be represented as a group-graph with k generators, having the group identity element as its root.

In a p-node tree the path distance between two arbitrary nodes is bounded by $\log_k p$. This implies that complete interprocessor communication in a tree-structured RPN may be achieved in polynomial-time.

The idea of the processing method is to assign tasks to cells in the network so as to mimic the operation of the nondeterministic machine to be simulated, as described before. A sequence σ_v in the nondeterministic machine, will correspond to a path going from the root to a node labeled v in the network. Since now complete communication among processors is possible, the search tree may be pruned very effectively, thus reducing the necessary number of active cells. By keeping only a polynomial number of best paths, optimal solutions will be generated in most cases. Otherwise, the solutions will be near-optimal, which is sufficient for many applications. Heuristic knowledge about the problem might be somewhat helpful in the definition of the merit function, but would not be required. The internode distance of a p-node tree built into a cubical structure is of order $p^{1/3}$. This limit on the speed-up seems less critical than the volume constraint.

A Methodology

Preliminaries

We introduce a general formulation in the propositional calculus that permits the examination of the dynamical behavior of RPN. For simplicity, the approach is first illustrated through an example.

Let us consider the question of completeness for the case of a 1-dimensional, 2-state, scope-2, uniform polygenic TPN. Let c_1 and c_2 be two configurations stipulated by

$$c_1 = \overline{0}x_0x_1x_2x_3\ldots x_m\overline{0}$$
$$c_2 = \overline{0}y_1y_2y_3\ldots y_m\overline{0},$$

where $\overline{0}$ denotes a quiescent portion. The neighborhood index is given by NI = ((-1,0), (0,0)). We need to determine the smallest length m for which there is a c_2, such that no c_1 can be found that is a predecessor of c_2. If such an m exists, this will mean that the TPN is incomplete. Notice that c_1 can be given the indicated form without loss in generality; if $\overline{0}x_0\ldots x_m\ldots x_{m+k}\overline{0}$ is a predecessor of c_2, then so is c_1.

A local transformation on two variables a,b can be expressed as $a\bar{b}z_1 + \bar{a}bz_2 + abz_3$, where '+' denotes logical OR and '-' represents complementation. Without loss in generality we may let $y_1 = y_m = 1$. We now state the following set of Boolean equations, where y_j, j=2,...,m-1 are the independent variables and x_i, i=0,...,m, z_k,k=1,2,3 the unknowns.

$$0 = x_0z_2 \qquad (0)$$
$$1 = x_0\bar{x}_1z_1 + \bar{x}_0x_1z_2 + x_0x_1z_3 \qquad (2)$$
$$y_2 = x_1\bar{x}_2z_1 + \bar{x}_1x_2z_2 + x_1x_2z_3 \qquad (4)$$
$$\vdots \qquad \vdots$$
$$y_{m-1} = x_{m-2}\bar{x}_{m-1}z_1 + \bar{x}_{m-2}x_{m-1}z_2 + x_{m-2}x_{m-1}z_3 \qquad (5)$$
$$1 = x_{m-1}\bar{x}_mz_1 + \bar{x}_{m-1}x_mz_2 + x_{m-1}x_mz_3 \qquad (3)$$
$$0 = x_mz_1 \qquad (1)$$
$$0 = \bar{z}_1z_2z_3 \qquad (*)$$

The last condition (*) has been included to rule out the identity transformation. The system is first expressed in the form g(x,y,z) = 1, and the set of interpretations of y that make the equation unsatisfiable is then established from it [8]. For m=3 this set is empty, and for m=4 patterns $\overline{0}1101\overline{0}$ and $\overline{0}1011\overline{0}$ are obtained. By performing a recursion on m, a general expression g for each length m can be derived.

A propositional language that deals with nonbinary state alphabets in a similar way has been defined. The description is lengthy, and thus is not included in this paper. We said that a global transformation preserves property Q when a configuration has property Q if and only if its successor has property Q. Properties of interest, especially for pattern recognition applications, include connectedness, convexity, having a given Euler number, monotonic growth, monotonic convergence to a set of goal configurations. The questions of completeness, controllability and reproducibility, with the stipulation that they preserve a set of properties, can be analyzed with the procedure we have outlined. Time bounds can be derived by obtaining a general form g through a recursion on the number of steps. The independent variables in the set of equations are now given by the initial and goal configurations x and y and the unknown variables shall correspond to the local transformations ${}^e z$, where superindex e refers to the step number. The minimum e that makes the system of equations solvable for an arbitrary interpretation of x and y is the desired lower bound on the number of steps required in a trajectory.

By solving the corresponding equation $g(x,y,z) = 1$ an optimal trajectory between two given patterns x and y can be derived. Since the expression for g may be determined by recursion on m and e, as illustrated before, properties such as completeness and controllability can be proven by induction on m and e. Lower time bounds on trajectories are of interest both in the uniform and non-uniform modes of operation.

The practical importance of deriving timing constraints as a function of interconnection structure and instruction set size lies in that they provide the designer with guidelines in choosing a network that meets his requirements. These questions may be investigated in the presence of different kinds of cell failure and from their understanding fault-tolerant schemas developed.

Periodic Configurations in Uniform Machines

We explore the concept of periodicity for uniform RPN. A periodic fragment c_S, defined by set S, is a subconfiguration of c, $c_S\colon S \to A$, such that S is the largest set of cells $i \varepsilon P$ for which there exists a set of points D, called the shift index, that verifies

$$c(i) = c(i \cdot L(D))$$

with the stipulation that $i \cdot L(D) \varepsilon S$, where L(D) is an arbitrary linear combination of the elements in set D. A periodic fragment in Z^d is rectangular if $S = S_1 x S_2 x..x S_d$. The periodic index PI of a periodic fragment defined by region S, is given by the largest subset of cells in S such that no two cells i_1, i_2 are in the relation $c(i_1) = c(i_2 \cdot L(D))$. The periodic index PI defines the template that repeats itself, and the set S indicates the region of periodicity. A pattern may have several periodic fragments. The following lemmas illustrate the way periodicity evolves throughout time.

Lemma 1 Let M be a uniform polygenic RPN with neighborhood index NI. Let c_1 be a configuration in M with periodic fragment defined by region S_1, periodic index PI_1 and shift index D. If the set of transformations I is unrestricted, the periodic region will be reduced in a successor configuration to S_2, where the lower bound for S_2 is given by

$$S_2 = \{i\colon \; NI \cdot i \subseteq S_1\}$$

Proof. Let $W = \{i\colon NI \cdot i \subseteq S_1\}$. If $i \notin W$, then there exists a linear combination L(D) such that the states of the neighborhood of cells i and $i \cdot L(D)$ differ, i.e. $c_1(N(i,NI)) \neq c_1(N(i \cdot L(D),NI))$, since by definition of a periodic fragment $c_1(NI \cdot i) \neq c_1(NI \cdot i \cdot L(D))$ and $N(i,NI) = NI \cdot i$. Thus, there exists a successor configuration c_2 such that $c_2(i) \neq c_2(i \cdot L(D))$. Therefore $i \notin S_2$ and $S_2 \subseteq W$.

Conversely, let $i \varepsilon W$. Then $c_1(NI \cdot i) = c_1((NI \cdot i) \cdot L(D))$ and thus $c_1(N(i,NI)) = c_1(N(i \cdot L(D),NI))$. So $i \varepsilon S_2$ because of the constraints of uniformity and $W \subseteq S_2$. ✗

Note that Lemma 1 does not hold if the definition of periodicity provides for preoperation by L(D) instead of postoperation. This occurs in the case of a group-graph where commutativity does not apply, as can be verified in the above proof. Lemma 1 characterizes how a given periodic pattern progresses throughout time. The following Lemma 2 illustrates the role of the neighborhood index in that evolution. We say that a configuration has periodic constraint when the set of its successors is restricted (due to the uniform operation of the machine) as a result of certain periodicity present in it, in the sense that there is at least a configuration that it can not directly produce.

Lemma 2. Let c be a configuration in M with periodic fragment defined by region S, periodic index PI and shift index D. Configuration c has no periodic constraint originated by S if and only if there is no cell $i \varepsilon P$ such that $NI \cdot i \subseteq S$.

Proof. Since $NI \cdot i \not\subseteq S$ for any $i \varepsilon P$, $c(NI \cdot i) \neq c((NI \cdot i) \cdot L(D))$. Thus $c(N(i,NI)) \neq c(N(i \cdot L(D),NI))$ for any $i \varepsilon P$. Consequently, periodic fragment c_S has no periodic constraint. ✗

The more periodic a configuration is, the longer it takes to produce any other pattern from it. The theorem makes this precise.

Theorem 1. Let M be a uniform polygenic RPN with neighborhood index NI. Let c be a configuration in M with periodic fragment defined by S, periodic index PI and shift index D. If there is no cell $i \varepsilon P$ such that $NI^j \cdot i \subseteq S$, where $X^2 = \{r \cdot s\colon r,s \varepsilon X\}$, then there is a configuration that can only be reached from c in at least j steps.

Proof. It follows from the previous lemmas. ✗

There are only $(\#A)^n$ different patterns of size n, given a state alphabet A. Thus, configurations with nonquiescent region of size at least $(\#A)^n$ will necessarily exhibit some periodic constraint since there will be cells in them with identical neighborhood states. The largest region of cells that can be free of periodic constraint, given a neighborhood index NI, is characterized in the following theorem.

Theorem 2. Let M be a uniform polygenic RPN with neighborhood index NI. The largest region of cells that may be free of periodic constraint is given by $NI^u.i$, where u is the largest integer such that NI^{u-1} will have no more than $(\#A)^n$ elements and i is the center of the region.

Proof. Every cell $j \epsilon NI^{u-1} \cdot i$ is such that $N(j,NI) \subset NI^u \cdot i$ since $NI \cdot NI^{u-1} \cdot i = NI^u \cdot i$. Any other region R with a different topology but the same number of cells as $NI^u \cdot i$ will be of no lower periodic constraint because its subregion $\{j: N(j, NI) \subset R\}$ will be of lower cardinality than $NI^{u-1} \cdot i$.

Structural Conditions for Reproducibility

The following question, of particular interest when a special-purpose machine has to be designed is studied. Given a set of configuration sequences a uniform RPN should generate, we ask what the necessary neighborhood index NI and set of admissible transformations I for the machine are.

Answers to this problem are developed by constructing algorithms which determine NI and I under various criteria of optimality. These procedures clarify the existing trade-offs between NI and I.

The concept of a prime neighborhood index proves to be useful. Let the configuration sequence $c_1, c_2, \ldots, c_v, \ldots$ be produced from initial configuration c_1 by a sequence of local transformations $z_1, z_2, \ldots, z_v, \ldots$ in I. Reproducibility is the ability of a machine to generate any member of a given class of configuration sequences. Let us consider the transition from c_v to c_{v+1} produced by the application of local transformation z_v. Neighborhood index NI_v is said to be prime at step v if

$$c_v(N(i_1, NI_v)) = c_v(N(i_2, NI_v))$$

$$\text{implies } c_{v+1}(i_1) = c_{v+1}(i_2),$$

$i_1, i_2 \ \epsilon \ P$ and there is no $NI \subset NI_v$ that fulfills that condition.

A method for deriving the set of prime neighborhood indices at each step v by using the propositional calculus has been developed, but it is not described here. A neighborhood index is said to be prime with respect to a given set of configuration sequences, if it is prime at each step of the sequences.

Let SNI be the set of prime neighborhood indices. Two criteria of optimality may be applied to determine an efficient interconnection structure. The aim of the first criterium is to minimize the neighborhood size. An optimal NI according to it is a NI ε SNI such that its cardinality is minimum.

In the second criterium, minimizing the cardinality of the set of admissible local transformations I is of basic concern. Now, neighborhood index NI is such that NI ε SNI and the set $\{z_v : z_v(c_v(NI \cdot i)) = c_{v+1}(i), \ i \epsilon P, \ v \epsilon V\}$ of local transformations implied by NI is of minimum cardinality.

There is a trade-off between NI and I because in general and neighborhood index that is of minimal cardinality will tend to imply a larger set of admissible transformations, and conversely. A way to find a NI that is optimal under both criteria is to define a combined merit function and then to select NI ε SNI that maximizes this function.

References

[1] H. Yamada, and S. Amoroso, "Tessellation automata," Inform. Contr. 14 (1969), pp. 229-317.

[2] H. Yamada and S. Amoroso, "A completeness problem for pattern generation in tessellation automata," J. Comput. System Sci. 4 (1970), pp. 137-176.

[3] H. Yamada, and S. Amoroso, "Structural and behavioral equivalences of tesselation automata," Inform. Contr. 18, 1 (1971), pp. 1-31.

[4] A. R. Smith III, "Two-dimensional formal languages and pattern recognition by cellular automata," Proc. 12th Annual Symp. on Switching and Autom. Theory (1971), pp. 144-152.

[5] A. R. Smith III, "Real-time language recognition by one-dimensional cellular automata," J. Comput. System Sci. 6, 3 (1972), pp. 233-253.

[6] J. I. Seiferas, "Observations on nondeterministic multidimensional iterative arrays," Proc. 6th ACM Annual Symp. on Theory of Computing (1974), 276-287.

[7] R. M. Karp, "Reducibility among combinatorial problems," In Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, Eds., Plenum Press (1972), pp. 85-103.

[8] M. Davio, and J. P. Deschamps, "Classes of solutions of Boolean equations," Philips Res. Reports 24 (1969), pp. 373-378.

A PROOF METHOD FOR CYCLIC PROGRAMS

Nissim Francez and Amir Pnueli
Weizmann Institute of Sciences
and
Tel-Aviv University
Israel

Abstract -- A formalism is developed for specifying and proving correct behaviour in time of cyclic (non terminating) programs. The statements use explicitly a time variable and program counters in order to specify correct response to external stimuli . The same technique is also shown to be applicable for concurrent programs, where each in turn is considered as an external environment acting on its associate.

I. Introduction

The computing activity generally [1,5] attributed to programs is the computation of some partial function over some domain. In other words, a program is an (algorithmic) realization of a given input - output relationship.

Hence a fundamental property of such functional programs is the Halting Property. A correct program should halt and produce as output the desired function of its inputs, while a cycling computation is either incorrect, or represents an undefined value of a partial function.

However, there are many programs whose essential role is to cycle forever, provided they respond correctly to incoming stimuli , or arising conditions. Examples of such programs are Operating Systems and, to some extent, Simulation programs, and even Artificial Intelligence programs.

Most of the effort invested so far in formalizing the notions of specifications and correctness of Programs has been directed towards functional Programs. We feel that cyclic, nonhalting (sometimes referred to also as continuous) programs require an extension of the current techniques, and deserve special attention, being an intrinsic part of very complex systems, where verification problems are most acute.

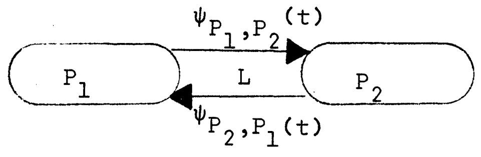
The essential difference between our approach and the usual "inductive assertions" method is that no longer can the execution be captured by snapshots at selected points in the program-text. The behaviour should be described as viewed at selected time instances, when significant events occur.

The programming model we propose to study in this paper is a continuously (ad infinitum) running program, which accepts external stimuli almost at random. Its behaviour is judged by its ability to respond correctly to those external events.

In order to describe the program's behaviour in time, we find it useful to introduce (into the proof only) the time variable t and the program location-counter variable π . While their explicit introduction in the case of functional programs might be considered a superfluous complication, it seems mandatory for cyclic programs.

Once a methodology for analyzing the behaviour in time of a cyclic program under external influence is developed, it can be utilized to analyze the joint behaviour of two (or more) concurrent cyclic programs, where each of them can be considered an external agent to the other.

Consider, for example, two concurrent programs, P_1 and P_2 , which communicate via an interface L (shared variables, signal lines etc.). Suppose we are able to characterize the desired communication between P_1 and P_2 by a predicate $\psi_{P_1,P_2}(t)$, and between P_2 and and P_1 by $\psi_{P_2,P_1}(t)$.



Suppose, further, that we were able to treat P_1 separately and show that provided its incoming signals satisfy $\psi_{P_2,P_1}(t)$, it will generate outgoing signals which satisfy $\psi_{P_1,P_2}(t)$. Assume P_2 can be treated similarly. Then, it is claimed that the correct joint behaviour of P_1, P_2 is ensured.

Usually, when proving $\psi_{P_2,P_1}(t)$ we do not need the full power of $\psi_{P_1,P_2}(t)$, and a weaker condition is sufficient. Correspondingly, we first prove that each program ensures a weaker commitment ϕ_{P_1,P_2} to its coprogram, and then, being ensured of such a condition in its turn, we can prove the full commitment ψ_{P_1,P_2} .

This method enables the partition of a problem involving two concurrent processes into two sub-problems involving each a single (sequential) process. (Sometimes, by symmetry, it suffices to treat only one of them.) As with other program verification methods, the requirement for formulating $\phi_{P_1,P_2}(t)$ and $\psi_{P_2,P_1}(t)$ explicitly, has intrinsic value of its own, forcing the programmer to account in a detailed manner his idea of the desired interface between the processes.

The importance of having a proof method encompassing the cases of continuous, nondeterministic and concurrent programs has also been underlined by Milner [7] .

The rest of the paper is composed as follows: In section II we introduce a detailed description of the programs model for a single (sequential) cyclic process. In section III we discuss in detail a case study, and specify and prove a scheduler for the "dining-philosophers" problem. In section IV we extend the model to include concurrent programs, and discuss in detail another case study, the "mutual exclusion" problem. Section V ends with a discussion.

II. A Model for cyclic (sequential) programs and their formal specifications

As our basic model we shall consider flow-charts without an ending (halting) node. The set of variables in a program is divided into three classes:

1. Event (input) variables, E_i .

These variables receive their values from some external source. The usual interpretation of such variables will be external requests or messages to which the system has to respond. The setting of these variables is completely unsynchronized with the program's control, and occur independently of the program counter state. Their values may be tested by predicates (tests) in the program.

2. Working (local) variables, Y_i .

These are the internal variables of the program, set by assignments in the program, and checked by tests in the program.

3. Status (output) variables, S_i .

These are variables through which the program contacts the external agent. They are set by assignments in the program. Usually, they will be interpreted as granting of requests, acknowledging a message or any other signal to the external agent.

$\bar{E}$, $\bar{Y}$, $\bar{S}$ will denote the vectors of Event variables, local variables and status variables, respectively.

Usually, a cyclic program is correct if, whenever a certain configuration appears in the Event variables, the program will eventually set a response configuration in the Status variables. In addition, it may be required that certain contradictory configurations (such as an allocation of the same resource to two processes) never arise.

In order to be able to state formally such claims, we need an explicit reference to the running time variable, t . As long as our discussion is qualitative, we can with no loss of generality assume that t takes successive integer values, starting with 0 . By this assumption, nothing interesting happens between integral time points, while a single instruction may take several time units to execute.

The value of t will often be used to parametrize the other variables, e.g. $y(t)$, y_t which signifies the value of y just after the instance t , when all changes have settled to their steady value.

The time value will often appear quantified. We shall use the notation $\forall t(P(t))\cdot Q(t)$ as an abbreviation for $\forall t\,[P(t) \supset Q(t)]$, and $\exists t(P(t))\cdot Q(t)$ as an abbreviation for $\exists t\,[P(t) \wedge Q(t)]$. For example $\forall t(t \geq t_0)\cdot Q(t)$, or $\exists t(t_1 \leq t \leq t_2)\cdot Q(t)$.

We also found it useful to introduce an explicit location counter variable, π , whose value is the next instruction to be executed. Note that π points to the place between statements, before the next statement and after the last executed statement. π enables us to state assertions which are valid at a set of locations in the program.

Consider the following illustration of a typical time dependent construct. We may want to express the idea of a variable's value being changed, and define the predicate

$$\mathrm{Set}_t\ (x,e) \equiv x(t-1) \neq e \wedge x(t) = e \ ,$$

where x is any variable and e any value.

An extension is $\mathrm{Set}_t(P)$, where P is a predicate (possibly involving several variables) and t is an instance when it becomes true (due to some assignment at instance t to some variable occurring in P).

A specification for the correct behaviour of a cyclic program consists of two parts: ϕ and ψ .

We intentionally use the standard [1] notation used in functional programs terminology, the intent being that ϕ denotes the input ($\bar{E}$) condition guaranteed by the external agent; and ψ denotes the output ($\bar{S}$) condition which characterizes the behaviour of the program.

Thus, for a cyclic program to be correct, it is necessary and sufficient that, whenever the external agent behaves in accordance to ϕ , the program will respond according to ψ .

Following are some examples of typical constituents of ϕ .

$$\forall t[\mathrm{Set}_t(Q(\bar{E})) \supset P_t(\bar{S})] \qquad (*\text{-}1)$$

Such an assertion ensures that $Q(\bar{E})$ will not be set, unless the program allowed for it, or is ready for it. For example, if we interpret $Q(\bar{E})$ as a resource allocation request, the meaning of *-1 could be that no such request will be issued if the resource is already allocated to the same requesting process.

$$\forall t_1,t_2(t_1 \leq t_2)\cdot[\mathrm{Set}_{t_1}(Q(\bar{E})) \wedge \forall t(t_1 \leq t \leq t_2)\cdot \sim P_t(\bar{S}) \supset Q_{t_2}(\bar{E})]$$

$$(*\text{-}2)$$

Such an assertion means that once a request is made by setting an event $Q(\bar{E})$, the requesting agent will not change its mind and reset the event, until the program responded to (or acknowledged) the request.

$$\forall t_1\ \exists t_2(t_2 \geq t_1)\cdot[P_{t_1}(\bar{S}) \supset \mathrm{Set}_{t_2}(Q(\bar{E}))] \ ,$$
$$\text{or}\ \ EV_1\ (P(\bar{S}),\ \mathrm{Set}\ (Q(\bar{E}))) \qquad (*\text{-}3)$$

We shall use the abbreviation $EV_1(P,Q)$ for $\forall t_1 \exists t_2\ (t_2 \geq t_1)\cdot[P_{t_1} \supset Q_{t_2}]$. This can be used to require, for example, that once a resource has been granted to an outside user, this user will eventually release the resource and notify the program by setting $Q(\bar{E})$.

*-1 to *-3 are not necessarily imposed on each event, but only as appropriate to the case. For example, if in *-3 , $Q(\bar{E})$ denotes a request rather than a release, we do not require that the agent ultimately asks for any particular resource.

Next, we give examples of some typical ψ-assertions.

$EV_1(\mathrm{Set}(Q(\bar{E})$, $\mathrm{Set}\ (P(\bar{S})))$, or, more fully,

$$\forall t_1 \exists t_2 (t_2 \geq t_1) \cdot [\mathrm{Set}_{t_1}(Q(\bar{E})) \supset \mathrm{Set}_{t_2}(P(\bar{S}))] \quad (*\text{-}4)$$

This assertion states that, eventually, every event will be properly handled. For example, every request will be granted after a finite delay.

$$\forall t \sim W_t(\bar{S}) \quad (*\text{-}5)$$

This assertion assures that some global ("eternal") limitation is never violated. This handles cases like conflicting requests, e.g. two simultaneous requests for the same resource, which should not be allocated to two users simultaneously.

$$\forall t [\mathrm{Set}_t(P(\bar{S})) \supset Q_t(\bar{E})] \quad (*\text{-}6)$$

This assertion states that no $P(\bar{S})$ happens unless explicitly requested by an event $Q(\bar{E})$. For example no resource is granted if not requested.

To summarize the notion of correctness of a cyclic (sequential) program we present the following definition:

Definition: Let P be a cyclic program, ϕ and ψ are explicitly time dependent predicates. The program P will be called correct w.r.t. ϕ and ψ if, whenever an external agent ensured a behaviour consistent with ϕ, the program P will respond in a behaviour ($\bar{S}$ -variation with time) satisfying ψ.

The concept of correctness defined above actually corresponds to what Manna [1] calls total correctness. As we shall see in the sequel, when proving correctness, we do not separate the proof into partial correctness and some analog of termination. Rather, we always show that control does follow some path in the flow-chart, and some assertion relates the starting and ending state-vectors. Thus, in order to prove a statement of the type $EV_1(P,Q)$, we have first to locate all the possible places where P might have become true. We then proceed to isolate and trace significant events which ultimately lead to Q becoming true. The passage from one such intermediate event to its successor is proved based either on inspection of the program behaviour (symbolic execution), or relying on some external commitment ϕ.

We want to stress that our model is qualitative in its nature in that the general statements to be proven are about ultimate proper response, where nothing is said about the length of the delay.

In a more quantitative treatment one will possibly have to define an "Eventually" predicate with some concrete time bound on the delay, i.e.

$$EV_1^*(P,Q): \exists N \forall t \exists t' (t \leq t' \leq t+N) \cdot P(t) \supset Q(t')$$

which will limit the eventual occurrence of Q within a certain predetermined period. This will make the use of such a commitment very convenient, but will be harder to prove as a commitment to others [8].

III. FIRST CASE STUDY: the Dining philosophers:

III.1 Informal presentation:

The example is based on a problem stated and solved by Dijkstra [2].

Five philosophers are seated around a round table, each one having a plate. Between every two neighbour philosophers is a fork. The eternal life of each philosopher alternates between the activities of eating and thinking. Both actions of eating and thinking are known to terminate after a finite amount of time (strictly positive). For performing his "eat" action, each philosopher needs exactly 2 forks. As a consequence, no two neighbour philosophers may eat simultaneously. When starting to think, the philosopher frees his forks. All the philosophers act completely asynchronously. The problem is to devise a system for which the philosophers act as external agents, requesting the allocation of forks and freeing them. A correct system should ensure that each request will be granted eventually.

Dijkstra's solution is by means of a synchronization via semaphores. We present here a sequential cyclic program (figure 1), which serves as a scheduler.

The following variables are used:

R[1:n] - An array of external-event-variables. R[j]=e means an Eat-request issued by philosopher j. Actually, this is a request for resource (forks!) allocation. R[j]=t means a Think-request. (It should not be confused with time-value). This in turn is a notification of resource release.

S[1:n] - An array of Status variables. S[j]=e means philosopher j is granted permission to eat; S[j]=t means philosopher j is allowed to think. We may assume that a philosopher j wishing to change his state (from e to t or vice versa) will issue a request through R[j] and sit watching S[j] until it switches to his desired new state, at which point he will immediately enter this state.

q - A queue variable, capable of holding indices of philosophers. The notation hq, tq, q•t means, respectively: first element of the queue, rest of the queue, insertion at the end of the queue. The predicate $q=\Lambda$ is true if q is an empty queue. $\oplus$, $\ominus$ represent addition and subtraction modulo n in the range 1,...,n.

The program has a main cycle consisting of three loops L_1, L_2, L_3. L_1 searches pending t-requests, which are immediately granted and recorded in the corresponding S-entry. Obviously, L_1 is executed n times per cycle.

L_2 looks for new e-requests. For any such request, the corresponding index is entered at the end of the queue. L_2 is also executed n times per cycle.

L_3 checks whether the e-request of the top element of the queue (if such exists) can be granted. If possible, it is taken out of the queue and the next element of the queue is tried. Otherwise, a new cycle starts. A request may be granted if the needed resources are free.

The idea behind the operation is most simple. Each philosopher in the queue waits until all philosophers placed above him are served. The one on the top will be served after his neighbours finish eating and release their forks. Once finished, they cannot interfere anymore, since any subsequent request will be placed at the end of the queue, beyond the waiting one. Thus, eventual service is guaranteed.

III.2 The Correctness proof:

In this section we outline a formal proof of the correctness, using the ideas discussed before.

III.2.1 Specifications:

First, we give the specifications:

$$\forall i(1\le i\le n)\cdot\forall t \begin{cases} 1.\ \ \mathrm{Set}_t(R[i],e) \supset S_t[i] = t \\ 2.\ \ \mathrm{Set}_t(R[i],t) \supset S_t[i] = e \end{cases} \qquad (\phi\text{-}1)$$

This ϕ -assertion establishes the interpretation of $S=e$ as the eating state, and $S=t$ as thinking state. It guarantees that a request will be issued only from an "opposite" state.

$$\forall i(1\le i\le n)\cdot\forall t\exists t'(t'\ge t)\cdot[S_t[i]=e\supset \mathrm{Set}_{t'}(R[i],t)] \qquad (\phi\text{-}2)$$

This establishes the fact that every "eat" action lasts only a finite period of time. The corresponding fact about the "think" action is not used in the proof, and so not stated in the specifications.

Next, we state the ψ-assertions.

$$\forall i(1\le i\le n)\cdot\forall t\exists t'(t'\ge t)\cdot \begin{cases} 1.\mathrm{Set}_t(R[i],t)\supset \mathrm{Set}_{t'}(S[i],t) \\ 2.\mathrm{Set}_t(R[i],e)\supset \mathrm{Set}_{t'}(S[i],e) \end{cases} \qquad (\psi\text{-}1)$$

This assertion states the requirement, that eventually every request will be granted.

$$\forall i(1\le i\le n)\cdot\forall t \sim [S_t[i]=S_t[i\oplus 1]=e] \qquad (\psi\text{-}2)$$

This describes the requirement that no two neighbour philosophers eat simultaneously.

$$\forall i(1\le i\le n)\cdot\forall t \begin{cases} 1.\ \ \mathrm{Set}_t(S[i],t) \supset R_t[i] = t \\ 2.\ \ \mathrm{Set}_t(S[i],e) \supset R_t[i] = e \end{cases} \qquad (\psi\text{-}3)$$

This means that the status is not changed, unless requested. No one is forced to eat (or think) against his will.

III.2.2 Proof of ψ-1.1 :

Let i_o and t_o be such that $\mathrm{Set}_{t_o}(R[i_o],t)$ holds. From ϕ-1.2 we have that $S_{t_o}[i_o]=e$. We perform a case analysis according to the value of $\pi(t_o)$, the program counter.

a. $\pi(t_o) \in L_1 \wedge i_{t_o} < i_o$(or, $i_{t_o} = i_o \wedge \pi(t_o)=\alpha_1$):

This case occurs when the setting of $R[i_o]$ happens while the program is within L_1 , and has not yet tested $R[i_o]$. By induction on the value of i , we have that for some $t_1 \ge t_o$ the following holds:

$$\pi(t_1)=\alpha_1 \wedge R_{t_1}[i_o]=t \wedge S_{t_1}[i_o]=e \wedge i_{t_1}=i_o$$

$S[i_o]=e$ remains[a] true since we do not pass through any assignment to S . $R[i_o]=t$ remains true by ϕ-1.1 .

Now the test α_1 is satisfied, and $\mathrm{Set}(S[i_o],t)$ occurs, which proves the claim ψ-1.1 for case a .

b. $\pi(t_o) \in L_3$:

This is the case that $R[i_o]$ was set to t while control was the in L_3 loop.

Let $\ell = |q_{t_o}|$ (the length of q). By induction on ℓ we get that, for some $t_2 \ge t_o$, the following holds:

$$\pi(t_2)=\alpha_o \wedge R_{t_2}[i_o]=t \wedge S_{t_2}[i_o]=e$$

$S[i_o]$ did not change because, by lemma $\bar{Q}$, $i_o \notin q$ for $t_o\le t\le t_2$, and we do not pass E with $i=i_o$. $R[i_o]=t$ remains true by ϕ-1.1 .

At t_2+1 we have $i=1 \wedge \pi=\alpha_1$, and the rest of the proof is like case a .

Lemma $\bar{Q}$:$\forall i(1\le i\le n).\forall t[i\in q_t \supset R_t[i]=e \wedge S_t[i]=t]$ (to be proved later)

c. $\pi(t_o) \in L_2$:

By induction on i , we get for some $t_3 \ge t_o$ that

$$\pi(t_3) = \alpha_3 \wedge R_{t_3}[i_o] = t \wedge S_{t_3}[i_o] = e$$

where the invariance follows as before. The rest of the proof is like case b .

d. $\pi(t_o) \in L_1 \wedge i_{t_o} > i_o$ (or, $i_{t_o} = i_o \wedge \pi_{t_o} \ne \alpha_1$)

As in case a , we get for some $t_4 \ge t_o$ that

$$\pi(t_4) = \alpha_2 \wedge R_{t_4}[i_o] = t \wedge S_t[i_o] = e$$

and the rest of the proof like in case c .

Q.E.D. ψ-1.1

III.2.3 roof of ψ-1.2 :

Let i_o and t_o be such that $\mathrm{Set}_{t_o}(R[i_o],e)$. From ϕ-1.1 we have $S_{t_o}[i_o]=t$. First, we show that $i_o \notin q_{t_o}$. For, assume $i_o \in q_{t_o}$. From $\mathrm{Set}_{t_o}(R[i_o],e)$ we know that $R_{t_o-1}[i_o] = t$, and so, by lemma $\bar{Q}$, $i_o \notin q_{t_o-1}$. Thus, $\mathrm{Set}_{t_o}(i_o \in q)$ is true. It follows that $\pi(t_o-1) = \alpha_2 \wedge i_{t_o}=i_o$, and the test yields the value T , which is a contradiction to

(a) Actually, this step in the proof needs further justification in more formal proof, to show that control does not leave L_1 until t_1 .

$R_{t_o-1}[i_o] = t$.

So, we have $R_{t_o}[i_o] = e \wedge S_{t_o}[i_o] = t \wedge i_o \notin q_{t_o}$

Now, we split the proof according to the possible values of π , to prove that

$$\exists t^*(t^*\geq t_o)\cdot[\pi(t^*)=\alpha_2 \wedge R_{t^*}[i_o]=e \wedge S_{t^*}[i_o]= t \wedge i_o\notin q_{t^*} \wedge i_{t^*}=i_o] \qquad (*-6)$$

i.e. that at a later moment we will be just on the verge of putting i_o into the queue.

a. $\pi(t_o) \in L_2 \wedge i_{t_o} < i_o$ (or, $i_{t_o}=i_o \wedge \pi_{t_o}=\alpha_2$, which means (*-6) is immediately true).
The claim follows directly by induction over i .
$S[i_o]=t$ remains true, for we do not pass through any assignment to S .

$R[i_o]=e$ remains true by ϕ-1.2 .

b. $\pi(t_o) \in L_1$:

By induction over i we prove that $\exists t_1(t_1\geq t_o)$ s.t.

$$\pi(t_1) = \alpha_2 \wedge R_{t_1}[i_o] = e \wedge S_{t_1}[i_o] = t \wedge i_o \notin q_{t_1} \wedge i_{t_1} = 1$$

$S[i_o]=t$ remains true, since we do not pass[b] through the assignment to S with $i=i_o$, for the test α_1 fails for i_o . $R[i_o]=e$ remains true by ϕ-1.2 .
The rest of the proof is like in case a .

c. $\pi(t_o) \in L_3$:

Let $\ell=|q_{t_o}| \geq 0$. By induction over i , we prove that $\exists t_2(t_2\geq t_o)$ s.t.

$$\pi(t_2) = \alpha_o \wedge R_{t_2}[i_o] = e \wedge S_{t_2}[i_o] = t \wedge i_o \notin q_{t_2}$$

$i_o \notin q$ remains true, because there is no addition to q in L_3 . $S[i_o]=t$ remains true, since we skip the assignment to S , because $i_o \notin q$. $R[i_o]=e$ remains true by ϕ-1.2 .
The rest of the proof is like in case b .

d. $\pi(t_o) \in L_2 \wedge i_{t_o} > i_o$ (or $i_{t_o}=i_o$ and $\pi \neq \alpha_2$).
By induction on i we get $\exists t_4(t_4\geq t_o)$ s.t.

$$\pi(t_4) = \alpha_3 \wedge R_{t_4}[i_o] = e \wedge S_{t_4}[i_o] = t \wedge i_o \notin q_{t_4}$$

$i_o \notin q$ remains true, since $i > i_o$ implies that we do not pass the addition to q with $i=i_o$.
The other two invariants are justified as before.
The rest of the proof is like in case c .

(b) Even if we have passed, still the new value would be t .

This finishes the proof of (*-6). Now the test will succeed, and i_o will be added to q , and by induction over i we have, for some $\bar{t}$ ($\bar{t}\geq t^* \geq t_o$) (this will be the end of the current loop in which we put i_o into q),

$$\pi(\bar{t}) = \alpha_3 \wedge i_o \in q_{\bar{t}} \wedge R_{\bar{t}}[i_o] = e \wedge S_{\bar{t}}[i_o] = t \qquad (*-7)$$

Now, we prove that $\exists t'(t'\geq\bar{t})$ s.t.

$$\pi(t') = E \wedge i_{t'} = i_o$$

which establishes ψ-1.2. From $i_o \in q$ it follows that there exist words x,z over {1,2,...,n} (possibly empty), s.t.

$$q_{\bar{t}} = z\cdot i_o\cdot x$$

The claim follows from the following lemma Q .

Lemma Q:

$$\forall x,z\ \forall i_o(1\leq i_o\leq n)\cdot\forall t\ \exists t'(t'\geq t)\cdot[q_t=z\cdot i_o\cdot x \supset \pi_{t'}=E \wedge i_{t'}=i_o \wedge q_{t'}=x\cdot y]$$

where $i_o \notin y$,

i.e. for each index which is currently in the queue, there will be a time in which it is taken out of the queue and granted its e state.
Proof: First, we prove another lemma, Q_o .

Lemma Q_o :

$$\forall x\forall i_o(1\leq i_o\leq n)\cdot\forall t\exists t'(t'\geq t)\cdot\exists y[\pi_t=\alpha_3\wedge q_t=i_o\cdot x\supset \pi(t')=E\wedge i_{t'}=i_o\wedge q_{t'}=x\cdot y]$$

where $i_o \notin x$,

i.e. Each index which is currently at the top of the queue will eventually be taken out.
Proof of Q_o : Let $\bar{t}$ be such that $\pi(\bar{t})= \alpha_3 \wedge q_{\bar{t}} = i_o\cdot x$ holds. By lemma $\bar{Q}$ we have $R_{\bar{t}}[i_o] = e \wedge S_{\bar{t}}[i_o] = t$. Since $q_{\bar{t}} = i_o\cdot x \supset q_{\bar{t}} \neq \Lambda$, control reaches γ with $i=i_o$. We distinguish between four possible subcases at t* , the time of arrival to γ .

1) $S[i_o \oplus 1] = S[i_o \ominus 1] = t$
2) $S[i_o \oplus 1] = e \wedge S[i_o \ominus 1] = t$
3) $S[i_o \oplus 1] = t \wedge S[i_o \ominus 1] = e$
4) $S[i_o \oplus 1] = S[i_o \ominus 1] = e$

1. In this case, the test γ yields immediately the value T , and control reaches E , which implies the claim of Q_o .
2. By ϕ-2 there exists[c] a $t_1(t_1\geq t^*)$ s.t.

(c) We take the least t_1

$\mathrm{Set}_{t_1}(R[i_o \oplus 1],t)$ is true. Thus, $i_o \oplus 1 \notin q_{t_1}$ (i.e. $i_o \oplus 1 \notin x$). During the period $t^* \leq t \leq t_1$: $S_t[i_o]=t$ remains true, for the test γ fails because $S[i_o \oplus 1] = e$, and we do not pass E.

For the same reason, no element is removed from the queue, but some elements were possibly added. So, $q = i_o \cdot x \cdot y_1$ for some y_1 (possibly Λ). $i_o \notin y_1$, because of the test before entering the insertion point, Q. Also, $R_t[i_o]=e$ remains true by ϕ-1.2.

Now, applying ψ-1 to t_1 and i_o, we get that there exists some(d) $t_2 (t_2 \geq t_1)$ s.t. $\mathrm{Set}_{t_2}(S[i_o \oplus 1],t)$ is true.

During the period $t_1 \leq t \leq t_2$: The test at γ still fails for $i_o \oplus 1$, and we do not pass through E. Thus, $S_t[i_o]=t$ remains true, and $q_{t_2}=i_o\, x \cdot y_1 \cdot y_2$ for some (possibly empty) y_2, s.t. $i_o \notin y_2$. Also, by ϕ-1.2, $R[i_o]=e$ remains true.

Now, at t_2 we have $S_{t_2}[i_o \oplus 1]=t$. Also, for the period $t^* \leq t \leq t_2$, $S_t[i_o \ominus 1]=t$ remains true. For, there are two possibilities:

1. $\mathrm{Set}(R[i_o \ominus 1],e)$ did not happen, and so $\mathrm{Set}(S[i_o \ominus 1,e]$ also did not happen.
2. $\mathrm{Set}(R[i_o \ominus 1],e)$ did happen. So, either $i_o \ominus 1$ did not enter the queue yet, or $i_o \ominus 1 \in y_1 \cdot y_2$, and is waiting for granting its new request.

Thus, at some t', $t' \geq t_2$, we have $\pi(t')=\gamma \wedge i_{t'}=i_o$. Again, for the same reason as just noted, $S_{t'}[i_o \oplus 1]=S_{t'}[i_o \ominus 1]=t$ still holds, the test succeeds and π reaches E with $i=i_o$. This proves Q_o for case 2.

3. This case is similar to case 2.
4. By ϕ-2, there exist(e) t_1,t_2 $(t_1,t_2 \geq t^*)$ s.t.

$$\mathrm{Set}_{t_1}[i_o \oplus 1,t] \wedge \mathrm{Set}_{t_2}[i_o \ominus 1],t]$$
$$(\text{assume } t_1 \leq t_2).$$

As before, this implies $i_o \oplus 1 \notin q_{t^*} \wedge i_o \ominus 1 \notin q_{t^*}$.

During the period $t^* \leq t \leq t_2$:

$S_t[i_o]=t$ remains true, for the test γ fails by assumption, and π does not pass through E.

For the same reason, no element is removed from the queue, and some elements were possibly added. Thus $q_{t_2} = i_o \cdot x \cdot y_1$ for some y_1 (possibly Λ). Again, $i_o \notin y_1$ because α_2.

(d) Again, we take the <u>least</u> t_2.
(e) Again, we consider the <u>least</u> t_1, t_2.

$R_t[i_o]=e$ remains true due to ϕ-1.2.

Now, we apply ψ-1 twice, to get $t_3,t_4 \geq t_2$ (assume $t_3 \leq t_4$) s.t.

$\mathrm{Set}_{t_3}(S[i_o \oplus 1],t) \wedge \mathrm{Set}_{t_4}[i_o \ominus 1],t]$ holds.

During the interval $t_2 \leq t \leq t_3$: The test γ still fails, and π does not pass through E. So, $S_t[i_o]=t$ remains true, and $q_{t_3}=i_o \cdot x \cdot y_1 \cdot y_2$ for some (possibly empty) y_2, s.t. $i_o \notin y_2$. Again, $R_t[i_o]=e$ remains true due to ϕ-1.2.

This means we reached a state as in case 3, which completes the proof. Q.E.D. Q_o

We may now proceed with the proof of Lemma Q.

Let $q_{\bar{t}} = z \cdot i_o \cdot x$, and $\ell = |z|$ (the length of z). We proceed by induction over ℓ.

For $\ell=0$, we have $z=\Lambda$, and we have exactly the claim of lemma Q_o.

Assume the lemma is true for z s.t. $0 \leq |z| < \ell$, and let z^* be such that $|z^*| = \ell > 0$. This means that $z^* = i^* \cdot z'$ for some i^*, and $|z'| < \ell$. Then,

$$q_{\bar{t}} = i^* \cdot z' \cdot i_o \cdot x .$$

By lemma $\bar{Q}$, we know that $R_{\bar{t}}[i^*]=e \wedge S[i^*]=t$. As before, by case analysis, we can show that for some $t^* \geq \bar{t}$,
$\pi(t^*) = \alpha_3 \wedge R_{t^*}[i^*] = e \wedge S_{t^*}[i^*] = t$, and we can apply lemma Q_o to i^*, to get for some $t'' \geq t^*$

$$q_{t''} = z' \cdot i_o \cdot x \cdot y$$

where $i_o \notin y$. And now, the proof follows by the induction hypothesis, since $|z'| < \ell$.

Q.E.D. Q

To finish the proof of ψ-1.2, we have to prove the lemma $\bar{Q}$, quoted above several times.

<u>Lemma $\bar{Q}$</u> : $\forall i(1 \leq i \leq n) \cdot \forall t[i \in q_t \supset R_t[i]=e \wedge S_t[i]=t]$

<u>Proof</u>: Let i_o, t_o be s.t. $i_o \in q_{t_o}$.

Let t^* be defined as $\max_{t \leq t_o} \mathrm{Set}_t(i_o \in q_t)$, i.e. t^* is the last time i_o entered the queue q. From the definition, it follows that

$$\forall t(t^* \leq t \leq t_o) \cdot [i_o \in q_t] . \qquad (*\text{-}8)$$

obviously, $t^* > 0$. Also, from the definition of t^* it follows that $\pi(t^*)=Q$, the only point of insertion into the queue.

From this we know that

$$R_{t^*}[i_o] = e \wedge S_{t^*}[i_o] = t \qquad (*\text{-}9)$$

and we have to show the invariance of (*-9) during the interval $t^* \leq t \leq t_o$.

Assume that $\exists t_1 (t^* \le t_1 \le t_o)\cdot[Set_{t_1}(S[i_o],e)$.

This means that $\pi(t_1)=E$, the only point where S is assigned e . But, $\pi(t_1)=E$ also means $i_o \notin q_{t_1}$, in contradiction to (*-8) . Thus, we establish that

$$\forall r(t^* \le t \le t_o)\cdot[S_t[i_o]=t]$$

Now, $\forall t(t^* \le t \le t_o)\cdot[R_t[i_o]=e]$ follows immediately by ϕ-1.2.

Q.E.D. Lemma $\bar{Q}$
Q.E.D. ψ-1.

III.2.4.1 <u>Proof of ψ-2</u>:

The proof of this part of ψ is easy, because in this program we have an explicit test to this effect.

Assume that there exist(f) i_o,t_o , such that $Set_{t_o}(S[i_o] = S[i_o \oplus 1]=e)$. (*-10)
Obviously, $t_o > 0$.

Without loss of generality, assume $S_{t_o-1}[i_o]=t \wedge S_{t_o-1}[i_o \oplus 1]=e$. It follows that $\pi(t_o) = \delta \wedge i_{t_o}=i_o$, a contradiction to the test δ .

Q.E.D. ψ-2

III.2.4.2 <u>Proof of ψ-3:</u>

a. For ψ-3.1 the result is immediate, since $Set_t(S[i_o],t)$ implies $\pi(t)=T$, and hence, the test α_1 was true, which implies $R_t[i_o]=t$.

b. For ψ-3.2, assume $Set_t(S[i_o],e)$. This implies that $\pi(t)=E$. But, $i_o=i_t=h\ q_t$, which means $i_o \in q_t$, and the result follows by lemma Q.

Q.E.D. ψ-3

This completes the proof of all ψ-assertions, and establishes the correctness of the program.

IV. Concurrent cyclic programs

IV.1 In our second case study, we depart from the previously described non-deterministic model of cyclic programs, to consider <u>concurrent cyclic programs</u>, where the concurrency is explicit, instead of its implicit nature in the previous model. It is interesting to observe that the same modes of correctness-definition and correctness-proof apply also in this case.

There is a methodological novelty in the analysis of concurrent programs proposed here, which was enabled by the introduction of time into the specifications of programs. The discussion of the correctness **of concurrent programs** in the literature considers a pair P_1 , P_2 of such programs as a single complex entity. When describing such a system as a process of transitions between states, a state consists of constituents determined by both P_1 and P_2 e.g. the values of the variables in both programs. (See [9] for such analysis.)

We suggest a viewpoint of concurrent programs, which isolates each program <u>as if it were acting alone, and condenses the effect of the second program in a ϕ-assertion</u>, on which it can count while computing. Thus, ϕ_1 will describe the commitment of P_2 in behaviour in time, which enables P_1 to achieve its own correct behaviour in time, ψ_1 , and likewise for ϕ_2 , ψ_2 . Also, there is a part $\psi_{1,2}$, which prevents violating some global restriction on both P_1 and P_2 .

The advantage of this approach is that instead of dealing with a joint behaviour of two programs (whose complexity is the product of the complexities of those two programs) we decompose the proof into two proofs, dealing each with one cyclic (sequential) program, and two pairs of $<\phi, \psi>$ specifications. The complexity in this case is only the sum of the complexities of the two programs. We do reduce the problem of concurrent programs into two problems of (sequential) cyclic programs interacting each with an external agent, whose behaviour is given through the ϕ specification. Hence, in addition to the steps in the proof that were required in the case of a sequential cyclic program, where ϕ was taken for granted, in the reduction of concurrent programs we have another step in the proof, which establishes that the ϕ-assertions are indeed true.

This approach has also the great methodological benefit of forcing the programmer to state explicitly what is "the point" of the interaction between P_1 and P_2 , and thus understand better his own programs. Such an attitude may lead to a more constructive way of designing concurrent programs, in a similar way as the introduction of loop-invariance in functional programs.

We shall assume the following model for concurrent programs: P_1 and P_2 are cyclic flowcharts, with non-disjoint sets of variables. At each time instant t , either π_1 or π_2 move one step (either assignment or test). Also, each π_i is bound to move a step after some time. Nothing is known about the relative speed of π_1 and π_2 .

<u>Method of Proof</u>:

In order to prove correctness of a pair of interacting concurrent programs P_1, P_2 , w.r.t specifications $\phi_1,\psi_1,\phi_2,\psi_2,\psi_{1,2}$ the following has to be shown:

a. P_1 is correct w.r.t $<$<u>true</u>, $\phi_2>$.
   P_2 is correct w.r.t $<$<u>true</u>, $\phi_1>$.
b. P_1 is correct w.r.t $<\phi_1, \psi_1>$.
   P_2 is correct w.r.t $<\phi_2, \psi_2>$.
c. $\psi_{1,2}$ holds.

In step a., we prove (with no assumptions) that each P_i is capable of guaranteeing its ϕ commitment. Then, given those ϕ-commitments, we prove that the ψ-behaviour follows. These are one level higher than the ϕ-assertions. Inde-

(f) Again, consider the <u>least</u> t_o .

pendently, we prove at step c_4 some joint properties.

We must note that not every set of concurrent programs is amendable to this approach. Some concurrent programs may be too tightly interactive to allow easy decomposition of the type we are suggesting here.

IV.2 SECOND CASE STUDY - The Mutual Exclusion Problem

We consider a problem which, like the first one, was first described by Dijkstra [3], who relates the solution to Dekker.

P_1, P_2 are two cyclic concurrent programs, which enter from time to time into a critical section. It is desired that in a given instance of time, only one program may reside in its critical section. The problem is to design a synchronization between the two programs, which will prevent the simultaneous entry of P_1 and P_2 into their critical sections, and will postpone the decision in case of conflict for only a finite time, and ensure that every wish to enter a critical section will eventually be fulfilled.

The solution is reached by means of three shared variables, c_1, c_2 and "turn". c_i is boolean and expresses a wish of P_i to enter its critical section. c_i is changed only by P_i, but is available for inspection by its companion. turn $\in \{1,2\}$, determines which of the two processes is to give up in case of conflict. (Figure 2).

The strategy in case of conflict is the following: If turn = 1, P_1 has the right to insist on entering the critical section. Thus, it enters the loop L_1, where it waits until P_2 gives up for a while. On the other hand, P_2 sees it is its turn to give up, and enters the loop R_2, in which it waits until P_1 changes "turn", after exit from the critical section.

In case turn = 2 initially, we have a symmetric case.

Now, if we try to analyze P_1 by itself, we see that there are two assumptions on the behaviour of P_2, which fully describe the interaction.

1. If it is P_1's turn to insist on its right, P_2 will give up for long enough so that P_1 can accomplish its wish.

2. If it is P_2's turn to insist, then some time later the right to insist will turn to P_1.

Since P_1 and P_2 are symmetric, the same argument applies to the assumption of P_2 about P_1's behaviour. One has still to show that it is impossible for P_1, P_2 to enter their C.S's simultaneously. This follows easily from the tests α_3, β_3, being always reached with their own c variable equal to 0.

IV.3 The correctness proof:

IV.3.1 Formalized specifications:

We shall add few abbreviations of more complex time dependent predicates. Let P,Q,R represent predicates over the state-vectors. $Ev_2(P, Q, R)$ will mean the following:

$$\forall t(P(t))\cdot\exists t_1(t_1\geq t)$$
$$\cdot[\forall t_2(t\leq t_2\leq t_1)\cdot Q(t_2) \supset \mathrm{Set}_{t_1}(R)]$$

The meaning of this predicate is the following. Suppose P was true at some moment t, then, there exists a later moment, t_1, s.t R will be set to true at t_1, provided Q was continuously true in the interval $[t, t_1)$. Q will usually correspond to some signal one program sends to another, while R corresponds to a proper response. P corresponds to some initial condition that held when Q arose.

Note that EV_2 only guarantees the setting of R to true. Sometimes a stronger requirement is needed, to the effect of holding R true for a while, (until it is detected) provided Q continues to hold. This will be expressed by the predicate EV_3 (P, Q, R), which means

$$\forall t(P(t))\cdot\exists t_1(t_1\geq t)\cdot\forall t_2(t_2\geq t_1)$$
$$\cdot[\forall t_3(t\leq t_3\leq t_2)\cdot Q(t_3) \supset R(t_2)]$$

These predicates enable us to state conditional commitment of one program to another.

Now we state the specifications.

$$EV_3(c_2 = 0, (c_1 = 0 \wedge \text{turn} = 1), c_2 = 1) \qquad (\phi\text{-}1.1)$$

This assertion expresses P_2's commitment to give up its wish to access the C.S (c_2=1), if it is P_1's turn to insist (turn=1). It will do so for as long as P_1 needs it (c_1=0). All this hold in case of a conflict (c_2=0).

$$EV_2((c_2 = 0 \wedge \text{turn} = 2)\ ,\ c_1 = 1\ ,\ \text{turn} = 1) \quad (\phi\text{-}1.2)$$

This assertion expresses P_2's commitment to eventually pass the right to insist to P_1 if it had this right and used it (turn=2 ^ c_2=0), provided P_1 gave up its wish to access its C.S and "waits patiently" (c_1=1). Here we rely on the property of the variable "turn", which can be set back to 2 only by P_1 itself. So, we need not worry about turn=1 staying true for a time interval.

The ϕ-2 assertion is similar, and expresses P_2's expectations of P_1.

$$EV_3(c_1 = 0\ ,\ (c_2 = 0 \wedge \text{turn} = 2),\ c_1 = 1) (\phi\text{-}2.1)$$

$$EV_2((c_1 = 0 \wedge \text{turn} = 1)\ ,\ c_2 = 1\ ,\ \text{turn} = 2 (\phi\text{-}2.2)$$

Next, we specify the ψ-assertions.

$$EV_1(\pi_1 = \alpha_3\ ,\ \pi_1 = \alpha_5). \qquad (\psi\text{-}1)$$

Thus, it is claimed that if $\pi_1(t)=\alpha_3$ which means P_1 wishes to access its C.S , then eventually, at some $t' \geq t$, $\pi_1(t')=\alpha_5$, which means the wish was fulfilled.

$$EV_1(\pi_2 = \beta_3 , \pi_2 = \beta_5) . \quad (\psi\text{-}2)$$

In addition, we have the global requirement, that never will P_1 and P_2 access their C.S simultaneously, which is expressed by

$$\sim\exists t(\pi_1(t) = \alpha_5 \wedge \pi_2(t) = \beta_5) \quad (\psi_{1,2})$$

As mentioned in section IV-1 , every node in the flow-chart once reached, will be left eventually. An exception to this rule are nodes α_1 (and β_1) , in which P_1 (P_2) may wish to remain forever, never requesting access to C.S. This, of course, should not affect the correctness.

IV.3.2 Proof of the specifications:

We shall outline the various steps in the proof method mentioned above. Because of symmetry, only half of the propositions need proof.

a. P_1 is correct w.r.t <true, ϕ_2> .

We show that P_1 is capable of fulfilling its commitment to P_2 with no additional premises, i.e. independently of P_2's behaviour.

a-1: We prove first ϕ-2.2 .
Let t_o be such that

$$c_1(t_o) = 0 \wedge turn(t_o) = 1 \wedge c_2(t_o) = 1 \quad (T_o)$$

Hence, $\pi_1(t_o) \in \{\alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7\}$. We perform a case analysis.

a-1.1:
$\pi_1(t_o) \in \{\alpha_5, \alpha_7\}$. Clearly, for some t_1 , $t_1 \geq t$, $\pi_1(t_1)=\alpha_1$, which implies the claim.

a-1.2:
$\pi_1(t_o)=\alpha_3$. Since, $c_2(t_o)=1$ by (T_o) and remains so subsequently, π_1 reaches α_5 , and the case is reduced to a-1.1 .

a-1.3:
$\pi_1(t_o) = \alpha_4$. The test α_4 succeed by (T_o) , and $c_2=1$ still holds by the ϕ antecedent, and so the case reduces to a-1.2.

a-1.4:
$\pi_1(t_o) = \alpha_6$. π_1 reaches the test α_8 , which fails. Hence π_1 reaches α_3 with $c_1=0$, and again, the case reduces to a-1.2.

Q.E.D. ϕ-2.2

a-2: Next, we prove ϕ-2.1.
Let t_o be s.t

$$c_1(t_o) = 0 \wedge c_2(t_o) = 0 \wedge turn(t_o) = 2. \quad (S_o)$$

a-2.1: Assume the following S_1 holds:

$$\sim\exists t_o'(t_o' \geq t_o)\cdot[\pi_1(t_o') = \alpha_1 \vee \pi_1(t_o') = \alpha_8] , \quad (S_1)$$

from which follows

$$\exists t^*\forall\bar{t}(\bar{t} \geq t^*)\cdot[\pi_1(\bar{t}) \in L_1] , \quad (S_2)$$

which, in turn, implies

$$turn(\bar{t}) = 1 .$$

Hence, ϕ-2.2 is vacuously true, with t_1 chosen as t^*

a-2.2: Now, assume $\sim S_1$ holds, i.e.

$$\exists t_o'(t_o' \geq t_o)\cdot[\pi_1(t_o') = \alpha_1 \vee \pi_1(t_o') = \alpha_8] . \quad (S_3)$$

We check the two subcases separately.

a-2.2.1:
$\pi_1(t_o') = \alpha_8$. In this case, we may choose the t_1 of the EV_3 predicate as t_o' . Once at α_8 , π_1 remains in L_2 with $c_1=1$ as long as P_2 needs, i.e. as long as turn=2 $\wedge$ $c_2=0$ remains true, as required by the EV_3 predicate.

a-2.2.2:
$\pi_1(t_o') = \alpha_1$. Thus, $c_1(t_o') = 1$. Again, there are two subcases. The first one is when P_1 remains at $Rest_1$ forever, and thus $c_1=1$ remains true (since not changed in $Rest_1$). We may then choose the required t_1 as t_o' .
The second subcase occurs if π_1 reaches α_2 , and hence also α_3 . If $c_2 \neq 0$, the EV_3 predicate is vacuously true. Otherwise, since turn=2 remains true by the antecedent, π_1 reaches α_8 , and the case is reduced to a-2.2.1.

Q.E.D. ϕ-2.1

b. P_1 is correct w.r.t <ϕ_2, ψ_1> .

We show now that, assuming ϕ_2 is guaranteed by P_2, P_1 is able to satisfy ψ_1 .
Let t_o be such that $\pi_1(t_o)=\alpha_3$. Hence, $c_1(t_o)=0$. In the trivial subcase, where $c_2(t_o)=1$, i.e. there is no conflict between P_1 and P_2 , α_3 yields false, and π_1 reaches immediately α_5 , which proves the claim.

Thus, assume $c_1(t_o)=0 \wedge c_2(t_o)=0$. We distinguish two cases, according to the value of $turn(t_o)$.

b-1: $turn(t_o) = 1$.
From t_o onwards, $\pi_1 \in L_1$ holds, and no

variables are changed. Hence, by ϕ-1.1 (EV_3 predicate), there exists a t_1 which makes it true. Let $t_1 = t'_o$, and define t''_o as

$$t''_o = \min_{t'' \geq t'_o} \pi_1(t'') = \alpha_3 ,$$

i.e. the next time when P_1 will evaluate the test α_3 . According to ϕ-1.1, $c_2(t''_o)=1$, and π_1 will reach α_5 , as claimed.

b-2: turn(t_o) = 2 .

Thus, π_1 will reach L_2 with $c_1=1$ remaining ture. Then, by ϕ-1.2, at some t'_o , $t'_o \geq t_o$, turn(t'_o) = 1 will become true, and remain so. Hence, π_1 will reach α_3 with $c_2=0$, and the case reduces to case b-1 .

Q.E.D. (ψ_1)

c. $\psi_{1,2}$

Proof by reductio ad absurdum. Assume the existence of a t_o

such that $\pi_1(t_o) = \alpha_5 \wedge \pi_2(t_o) = \beta_5$

Let $t_1 = \max_{t \leq t_o} \pi_1(t) = \alpha_5$

$t_2 = \max_{t \leq t_o} \pi_2(t) = \beta_5$

and assume $t_1 \geq t_2$.

Thus $c_2(t_2)=0$. Since c_2 is not modified in the C.S we have also $c_2(t_1)=0$, which contradicts the assumption that α_3 fails at t_1.

Q.E.D. ($\psi_{1,2}$)

V. Discussion:

V-1 Comparison with related methods:

a. Actually, our method is an extension of Floyd's method of induction assertions, if applied to functional programs. The introduction of the implicit variables of time and program-pointer enables one to express the propositions which follow from Floyd's method.

Thus, if $\{Q_i\}$ is the set of inductive assertions of a given program at cut-points α_i , the following propositions have to hold in order to prove correctness:

$$\forall t[\pi(t) = \alpha_i \supset Q_i(\bar{Y}(t))]$$

$$\exists t[\pi(t) = \text{Halt}]$$

Of course, the introduction of implicit time and program-pointer variables complicates the formalism, and should be used only when necessary.

The main idea behind the inductive assertion method is induction over the path of computation. One shows that _if_ computation ever reaches α_i , then $Q_i(\bar{Y})$ holds. Yet, our approach is in a way an _"inverse approach"_, which exhibits another kind of induction, more related to Burstall's structural-induction. We show that _if_ so and so happens, then the computation _will eventually_ reach α_i .

b. Thus, our method also extends Burstall's method [4] . His notation

$$\alpha: \ Q(\bar{Y})$$

which means that currently computation is in the point α with Q true for the $\bar{Y}$'s at that moment, might of course be stated as

$$\pi(t) = \alpha \wedge Q(\bar{Y}(t))$$

Also, his more general statement

$$\alpha: \ Q(\bar{Y}) \supset \beta: \ P(\bar{Y})$$

may be translated as follows:

$$\forall t \exists t'(t' \geq t) \cdot [\pi(t)=\alpha \wedge Q(\bar{Y}(t)) \supset \pi(t')=\beta \wedge P(\bar{Y}(t'))]$$

Yet we allow more complex quantifications over time; Also, since we include non-determinism, we do not bind the Q's to "space" points α_i , but only to time points, since one does not know where control resides when an event occurs.

Thus, we may group together (during the proof) various points in the program, which are equivalent with respect to the occurrence of an event.

References

1.	Z. Manna:	- _Math. theory of computation_; McGraw-Hill, 1974.
2.	E.W. Dijkstra:	- "Hierarchical ordering of sequential processes"; in: _Operating systems techniques"_ C.A.R. Hoare, R.H. Perrott (eds.); Academic Press, 1972.
3.	E.W. Dijkstra:	- "Cooperating sequential processes"; in: _Programming Languages_; F. Genuys (ed.); Academic Press, 1968.
4.	R.M. Burstall:	- "Program Proving as Hand Simulation with A Little Induction"; in: _Information Processing_ 74; North-Holland 1974.
5.	R.W. Floyd:	- "Assigning meaning to programs"; in: T. Schwarz (ed.) _Math. Aspects of Computer Science_; A.M.S.; Providence, R.I., 1957.
6.	N. Francez:	- "The Analysis of Cyclic Programs"; Ph.d thesis; Dept. of Applied Mathematics, Weizmann Inst. of Sci.; to appear.
7.	R. Milner:	- "Processes: a Mathematical Model of Computing Agents"; in: _Proceedings of the Logic Colloquium Bristol_, 157-173.

7. R. Milner: (cont.) - North-Holland, 1973.
8. C.A.R. Hoare: - Private communication.
9. P. Gilbert and W.J. Chandler: - "Interference between Communicating Parallel Processes"; CACM 15, 6, 427-437, June 1972.

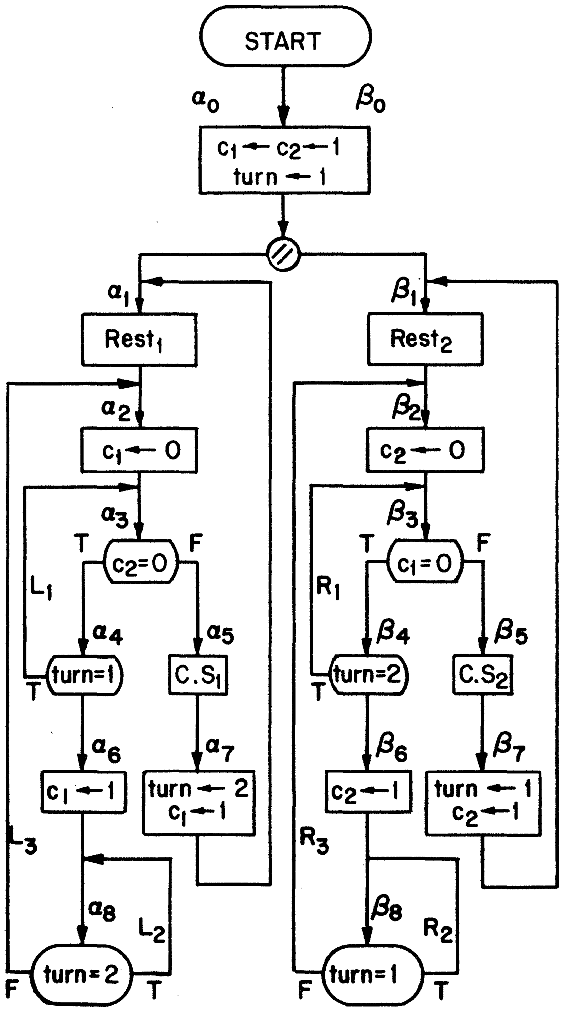


Fig. 2

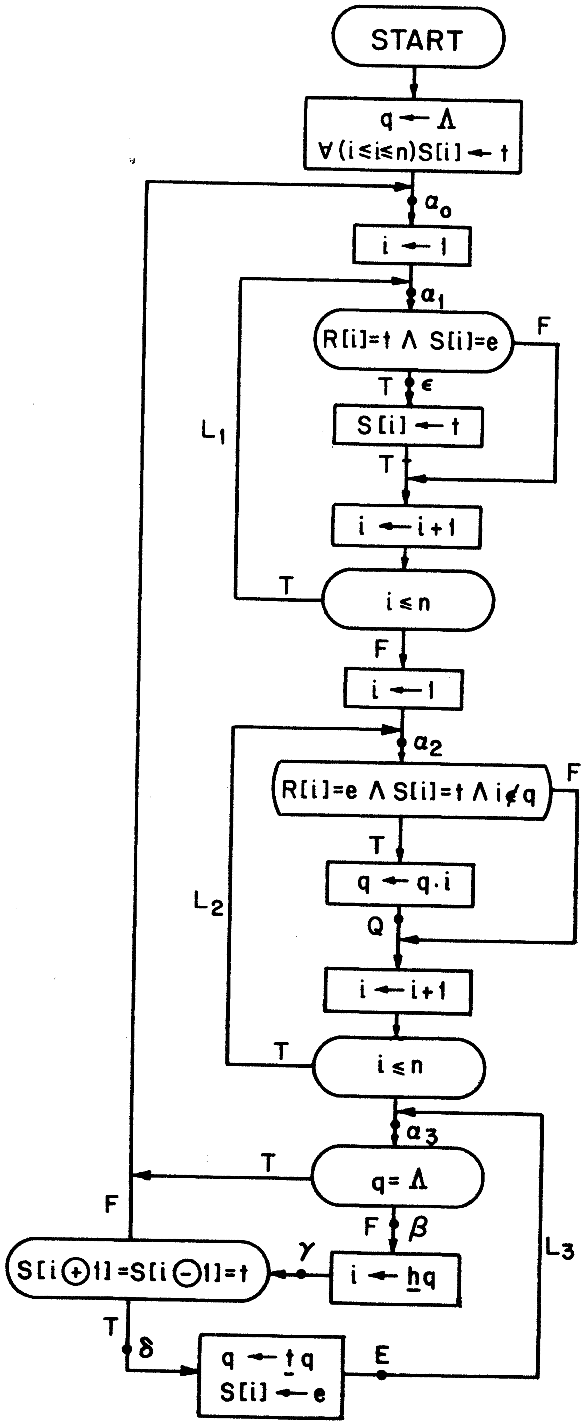


Fig. 1

ON DETERMINACY AND EQUIVALENCE OF PARALLEL PROGRAM SCHEMATA

Manilal Daya
Honeywell Information Systems, Inc.*
Billerica, Massachusetts 01821

Summary

A formal model of parallel computational schemata is the basis of this research. The specifications of this model have been influenced by various existing models [3],[4] and [6] so that the determinacy and equivalence problems for "repetition-free" [3] schemata can be investigated uniformly. A schema is repetition-free if it is both "free" and "liberal" [6]. The motivation for considering repetition-free schemata only is that if this hypothesis is excluded then most of the properties of interest are undecidable [5]. The results summarised here appear in full in [2].

Briefly, a parallel program schema consist of: a set of memory cells, a set of operations which depend on and may affect memory cell values, and a control specification which defines the flow. The schema model is related to the "realization" model in [4] with the following exceptions: (i) A halting state is explicitly introduced in the control. The "persistent and finite delay properties" associated with a computation [4], are excluded. (ii) During execution, there is a choice involved in selecting the next state from the current state.

A schema is: (a) 0-determinate (resp., Ω-determinate) iff for each interpretation if there exists a halting computation then all computations are halting with identical final memory cell values (resp., identical memory cell history [4]); and (b) ρ-determinate iff for each interpretation if there exists a halting computation then all computations are halting and syntactically equivalent[4].

The notion of progression introduced in [6] is generalised in the following way: A schema is progressive iff for any halting computation, at least one of the values written in memory by any operation executed during that computation, is either used by a succeeding operation encountered in that computation whose set of range cells is non-empty or remains unchanged until the end of the computation. It is shown that a repetition-free and progressive schema is 0-determinate iff it is Ω-determinate. Similar conclusion also holds true between 0-equivalence and Ω-equivalence.

A concept of minimality is introduced for repetition-free schemata. This concept is a generalisation of the notion of minimum state finite automata. It is shown that if a repetition-free schema has a minimal schema then it is unique up to an isomorphism (i.e., a renaming of the states).

The notion of minimal schemata is important because it implies code-optimization and "canonical-form" in some sense. As a result of this, it is shown that ρ-determinacy is decidable for a class $\mathcal{R}$ of repetition-free schemata. The class $\mathcal{R}$ is defined in a complex way based upon the notion of minimality; and membership in $\mathcal{R}$ is not known to be decidable. The corresponding "independent" [6] schemata defined from non-deterministic two-tape one-way automata [1] are also properly included in the class $\mathcal{R}$. (It can be also shown that the class of repetition-free, "resolvable" [4] schemata is also properly included in the class $\mathcal{R}$.) It is asserted that every conservative and ρ-determinate schema has a minimal schema. This assertion implies: (a) $\mathcal{R}$ is the class of finite state and repetition-free schemata; and (b) ρ-equivalence is decidable for the class of repetition-free and ρ-determinate schemata (which incidently also implies that the equivalence problem for n-tape one-way deterministic automata is decidable).

A notion of "Ω-reducibility" is introduced which means removal of redundant lossy operations. It is shown that a repetition-free schema S is Ω-determinate iff an Ω-reduced schema of S is Ω-determinate. The significance of this is that based on an assertion that repetition-free, Ω-reduced and Ω-determinate schema is ρ-determinate, it is shown if ρ-determinacy is decidable for repetition-free schemata then Ω-determinacy is also decidable. The result mentioned above can also be extended for 0-determinacy by including the progressive property mentioned earlier.

*This research was completed while the author was a graduate student at Harvard University - see [2].

References

[1] M.R. Bird, "The Equivalence Problem for Deterministic Two-tape Automata", J. Comput. Sci, Vol. 7, 1973, pp. 218-236.
[2] M. Daya, "A Study in Parallel Program Schemata", Ph.D. Thesis, Tech. Report TR 2-75, Center for Research in Computing Technology, Harvard University.
[3] R.M. Karp, R.E. Miller, "Parallel Program Schemata", J. Comput. Syst. Sci., Vol. 3, No. 2,1969, pp.147-195.
[4] R.M. Keller, "Parallel Program Schemata and Maximum Parallelism I & II", J. of the ACM, Vol. 20, Nos. 3&4, 1973.
[5] R.E. Miller, "Some Undecidable Problems for Parallel Program Schemata", SIAM J. Comput., Vol. 1,No. 1,1972,pp. 119-129.
[6] M.S. Paterson, "Equivalence Problems in a Model of Computation", Doctoral Thesis, Cambridge University, 1967; reissued as MIT Artificial Intelligence Lab. Memo. No. 1, 1970.

COORDINATION OF PARALLEL PROCESSES IN PL/1

HOWARD S. MODELL, M.S.
COMPUTER SCIENCE TEACHING ASSOCIATE

RONNIE G. WARD, Ph.D
ASSISTANT PROFESSOR IN COMPUTER SCIENCE

TED M. SPARR, Ph.D
ASSISTANT PROFESSOR IN COMPUTER SCIENCE
UNIVERSITY OF TEXAS AT ARLINGTON
ARLINGTON, TEXAS 76019

Abstract

This paper explains procedures to coordinate and control parallel processes in PL/1 under OS/370. The necessary additions to simulate Dijkstra P and V operations are discussed. Examples are provided to illustrate usage of these additions.

Introduction

Considerable interest has developed in multiple CPU architecture to solve distributed processing problems. The use of microprocessors has intensified this interest. The difficulties of direct program checkout on microprocessors has resulted in the frequent use of simulations on larger machines for initial phases of checkout. This simulation approach is even more desirable for more complex multiprogramming solutions. The multitasking capabilities of PL/1 [1] suggest that PL/1 should be an ideal vehicle for verifying multiple CPU algorithms targeted for smaller machines, as well as for implementing production programs. However, PL/1 does not provide sufficient control mechanisms for implementation of concurrent processes. The purpose of this paper is to present tools for use in simulations written in PL/1 that involve concurrent processes. These tools have been utilized extensively, and to our knowledge operate correctly. This paper does not apply to PL/1 implementations on machines which already provide synchronizing primatives via hardware.

Coordination Tools

A major source for problems when writing a program with concurrent tasks is coordinating the tasks with respect to their access and usage of resources (e.g. memory, I/O devices, etc.). As a general rule, it is desirable to restrict that access to only one task at any one time.

There have been several different algorithms or methods proposed as solutions to this coordination problem. One such solution is "Dekker's Algorithm" [2], which allows each task in turn to enter the critical region*. (See Figure 1) Another solution involves the use of the PRIORITY pseudo-variable in PL/1. Before entering its critical region, a task raises its own priority to a value greater than that of any other task in the program. Thus, when the CPU Manager next gives CPU time to any task of this job, the task with the highest priority will be given that time. After leaving the critical region, the task lowers its relative priority back to its original value. (See Figure 2)

Both of these solutions have disadvantages inherent to them that preclude our using them as general coordination methods. Dekker's algorithm, for example, "schedules"

----

* By "critical region" is meant that block of code which accesses or manipulates a resource that only one task at a time should be accessing or manipulating.

tasks to enter their critical region on a strictly cyclic basis. That is, if task number "i" is in the critical region now, task number " (i modulo n) + 1" [n = number of tasks in program] will be permitted to enter the critical region next. Although this strictly cyclic assignment of turns can be altered, there is still the potential that when task i releases the critical region, no other task has asked for it. The releasing task must arbitrarily pick another task, task "x", for the next turn. If task "x" does not need the critical region when TURN = 'x', then all the other tasks will be locked out. In addition, Dekker's algorithm is not easily adaptable to more than one critical region, and is not adaptable at all to a multiple CPU machine. Lastly, when a task is waiting to enter the critical region, it does this by executing a undesirable "spin-loop".

The use of the PRIORITY pseudo-variable also has its disadvantages, which while subtle, are none the less significant. When one task gets control of the critical region by increasing its relative priority, this necessarily implies that as long as that priority remains high, the CPU will remain "in the custody" of that task, and therefore, all the other tasks of that job will remain READY but non-executing. This includes tasks that are not trying to enter the critical region, which is wrong; the only tasks that should be suspended are those "desiring" to enter the critical region. This method also can not be adapted to a multiple - CPU machine.

A more suitable solution to the coordination problem is the method proposed by Dijkstra [3] which uses semaphores and two operators, P and V. A semaphore is either a boolean or integer variable, logically associated with the resource that needs coordinated access. Whenever a task "desires" access to the critical region/resource, it performs a P operation on the semaphore for that resource/region. If the region is unoccupied (i.e. the resource is not being used), then the semaphore is set to "occupied", and the task proceeds to enter the critical region. If the region is occupied, then the task performing the P operation is suspended. When a task leaves a critical region, it performs a V operation on the semaphore for the region/resource it controls. If there is any tasks suspended and waiting for that resource/region, one is selected and reactivated. If no tasks are waiting, then the semaphore is reset to "unoccupied". In either case, the task performing the V can continue execution.

There are at least two different ways of implementing P and V, one which involves actually creating and maintaining a queue for suspended tasks, and the other which involves essentially a test loop. We have already discussed the first implementation [4], in which we called the procedures PROCURE and LIBRATE. In several months of usage, in a variety of simulation programs, [5,6] the routines have performed adequately.

The second implementation is based on a different description of P/V [7]. In this version, when a task performs a P operation, if the resource is not available the task enters a test loop in which it WAITs for the resource to be free, at which time the availability of the resource is again checked. If it is still free, the task acquires it; else it WAITs again. (See Figure 3)

Both implementations have their advantages and disadvantages. The queueing method has the advantage that some queueing discipline other than straight FIFO is desired, (e.g. a scheme based on task priorities) it can be accomplished. The other method has the advantage that it is shorter (in terms of number of statements), and may be faster in execution since it is not performing queue manipulation. Thus the advantages of one implementation imply the disadvantages of the other. As far as we have been able to determine, there does not seem to be any significant difference in execution time of programs run using first one, then the other set of routines.

The key problem that had to be solved in order to implement these routines was the problem of making the P and V procedures act as indivisible operations. Another way to state this is to state that the procedures for P and V are themselves critical regions. The solution we devised is to write two ALC routines, ENQUEUE and DEQUEUE, which do nothing more than issue ENQ and DEQ macros [8] on a single resource. ENQ and DEQ are ALC MACRO instructions which act like P and V primatives, respectively. When a task wishes to access a resource, it issues an ENQ on that resource,
naming the resource and the name of a queue. If the resource is already held, the task is "entered" into the queue, and suspended. If the resource is free, the ENQueing task gets possession of it. When the task is finished with the resource, it releases it by issuing a DEQ macro, naming that same queue. If the queue is occupied, the first task within is dequeued and reactivated. The DEQueing task continues, regardless. The main body of the P and V procedures are "bracketed" by calls to these ALC routines, and only the first task to enter P or V gets to execute it; any "late-comers" are suspended until the executing task performs a DEQ, releasing the P or V procedure.

This approach to obtaining indivisibility parallels that developed by McGowen and Kelly [9]. They also use ALC ENQ/DEQ macros to bracket the critical region within REQUEST and RELEASE routines (similar to PROCURE and LIBRATE). Their implementation of ENQ/DEQ provides for mutual exclusion for only those tasks REQUESTing a particular resource. While in our implementation, only one task can execute PROCURE or LIBRATE at any one time, regardless of the resource. Their implementation provides for separate allocation and deallocation routines for each resource. In our first implementation [4], the same queueing strategy is used for all resources. In our second implementation, (see Figure 3) the environment is totally competitive; when a resource is freed, all tasks "queued" for it are released to try for it. Our routines, while less general, are somewhat simpler to understand and require less user initialization of special data structures.

While the exclusive use of a resource by a task is a common situation in concurrency simulation programs, it is not the only manner in which concurrent tasks can use resources. Provided that the resource is guarenteed to remain unaltered by tasks which access it (e.g. a read-only-dataset), there is no reason why more than one task cannot be accessing that resource simultaneously. That is, the tasks share the resource. The only restriction on this type of resource usage is that if the types of "users" are mixed (i.e. some who need exclusive access, and some who can share the resource), then the coordination routines need to be modified so that the type of access is taken into account when allowing a task to enter the critical region.

The criterion for entry into the critical region now becomes: if one task who can share the resource has control of it, other sharers can be allowed to enter the critical region at the same% time, while any exclusive-use task that requests entry to the region is suspended. On the other hand, if an exclusive-user is in the region, then everyone else requesting entry to the critical region is suspended.

We have implemented a pair of "shared access" coordination routines. We used the names PROCURS and LIBRATS to distinguish them from the exclusive-access-only routines. (See Figure 4)

It must be noted that with all four routines, PROCURE,
LIBRATE, PROCURS and LIBRATS, the user is responsible for using the routines properly. This entails the allocation and initialization of the data structures that the routines work on. For PROCURE and LIBRATE, the correct data structure is a PL/1 EVENT variable

for each resource for which coordination is desired, and each of these EVENTS must be initialized to "complete" (i.e. '1'B). PROCURS and LIBRATS, on the other hand, act upon a slightly more complicated structure. For each resource for which coordination is desired, the user must allocate a data structure composed of two EVENTs -- one for shared-access, and one for exclusive-access -- and a counter for the number of shared-access users who currently control the resource. Both events must be initialized to "complete", and the counter must be initially zero.

Regardless of which pair of routines is used, the programmer must not modify the data structures in any way. Once initialized, only the coordination routines are to operate upon those data structures. Also, the programmer is currently responsible for insuring that each PROCUR has a LIBRAT. (See [4] for suggestions on a possible solution to this problem.)

Applications Of Coordination Tools

In the ten months since the design and implementation of these coordination routines, they have been utilized quite extensively. They have seen particular exercise in two simulation projects carried out at UTA.

The initial project was carried out as a class assignment, and involved simulating a hypothetical operating system [5]. The problem was to write a simulation of an operating system which controlled two "card readers", two "line printers", a "drum" for secondary storage. There were also a "loader", "drivers" for each of the "I/O devices", and three "job initiators". "Input" and "Output" were allowed to occur concurrently, both within and without a "job". In one particular design, semaphores were associated with each "device", "driver", and other resource. PROCURE and LIBRATE were incorporated into two larger routines, P and V, which operated on counting semaphores (the semaphore can take on any integer value, with 0 as the threshold for queueing), and maintained a "future events list".

The second project was the subject of a graduate research project in Artificial Intelligence [6]. The research involved the design of routines for computer image feature extraction which used parallel processing to speed up the extraction process. The algorithm that was tested via the simulation involved dividing the digitized image being scanned into 3-point-by-3-point arrays, and initiating multiple tasks to analyze them, one task to one array. As there were several resources common to all tasks, the tasks had to be coordinated with respect to their access to those resources. This was accomplished by associating semaphores with each resource, and bracketting the code in which access occurred with PROCURE and LIBRATE.

Conclusions

The PROCURE, LIBRATE, PROCURS and LIBRATS routines described in this paper enable general purpose coordination of tasks in PL/1. Within the limits of our ability to accurately trace the execution behavior of the programs we have concluded that our implementations do properly coordinate concurrent tasks, giving both mutual exclusion and shared access to critical resources. The suitability of PL/1 for verification and implementation of multitasking algorithms is thereby enhanced.

```
In MAIN:
     DCL (C1,C2,TURN) BINARY(15);
     C1,C2 = 0; TURN = 1;

In task #i  (where i ~= j):
     Ci = 1;
     DO WHILE(Cj = 1);
       IF TURN~=i
         THEN DO;
              Ci = 0;
              DO WHILE(TURN~=i);
              END;
              Ci = 1;
              END;
     END;
/* critical section */
   Ci = 0;
   TURN = j;
/* remainder of task */
```

FIGURE 1. Dekker's Algorithm.

```
      PRIORITY = 100;
/*  statements which consititute */
/*       critical section        */
      PRIORITY = 0;
```

FIGURE 2. Using PRIORITY Psuedo-variable to obtain exclusive access.

```
PROCURE: PROCEDURE(SEM_ADDR)
              OPTIONS(REENTRANT);
  DCL SEM_ADDR POINTER,
      BINARY_SEM EVENT
                  BASED(SEM_ADDR),
      (COMPLETION) BUILTIN,
      (ENQUEUE,DEQUEUE) ENTRY
            OPTIONS(ASM INTER) EXT;
 CALL ENQUEUE;
 DO WHILE
  ( COMPLETION(SEM_ADDR->BINARY_SEM));
    CALL DEQUEUE;
    WAIT(SEM_ADDR->BINARY_SEM);
    CALL ENQUEUE;
 END; /* OF DO-WHILE */
 COMPLETION(SEM_ADDR->BINARY_SEM)='0'B;
 CALL DEQUEUE;
 RETURN;
 END PROCURE;


LIBRATE: PROCEDURE(SEM_ADDR)
      OPTIONS(REENTRANT);
DCL SEM_ADDR POINTER,
    BINARY_SEM EVENT BASED(SEM_ADDR);
DCL (COMPLETION) BUILTIN,
    (ENQUEUE,DEQUEUE) ENTRY
         OPTIONS(ASM INTER) EXT;
CALL ENQUEUE;
COMPLETION(SEM_ADDR->BINARY_SEM)='1'B;
CALL DEQUEUE;
RETURN;
END LIBRATE;
```

FIGURE 3. PROCURE AND LIBRATE

```
PROCURS : PROC(SEM_ADDR,REQ) OPTIONS(REENTRANT);

       DCL SEM_ADDR PTR,

           1 SEMAPHORE BASED(SEM_ADDR),

              2 EXCL EVENT, 2 SHR EVENT,

              2 #_SHR FIXED BIN(15),
           (COMPLETION) BUILTIN, REQ CHAR(1),
           (ENQUEUE,DEQUEUE) ENTRY OPTIONS(ASM INTER) EXT;
      CALL ENQUEUE;
      IF REQ = 'S'   /* I.E. IF USER WANTS SHARED ACCESS */
         THEN DO;   /* SHARED ACCESS OF RESOURCE */
              DO WHILE( COMPLETION(SEM_ADDR->EXCL));
              CALL DEQUEUE; WAIT(SEM_ADDR->EXCL); CALL ENQUEUE;
              END;  /* OF DO-WHILE */
              COMPLETION(SEM_ADDR->SHR) = '0'B;
              SEM_ADDR->#_SHR = SEM_ADDR->#_SHR + 1;
              CALL DEQUEUE; RETURN;
             END;  /* OF THEN DO */
         ELSE DO;   /* EXCLUSIVE ACCESS TO RESOURCE */
              DO WHILE(~COMPLETION(SEM_ADDR->EXCL) |
                       ~COMPLETION(SEM_ADDR->SHR));
                 CALL DEQUEUE;
                 WAIT(SEM_ADDR->EXCL,SEM_ADDR->SHR);
                 CALL ENQUEUE;
              END;  /* OF DO-WHILE */
              COMPLETION(SEM_ADDR->EXCL) = '0'B;
              CALL DEQUEUE; RETURN;
              END;  /* OF ELSE DO */
END PROCURS;


LIBRATS: PROC(SEM_ADDR) OPTIONS(REENTRANT);
       DCL SEM_ADDR PTR,
           1 SEMAPHORE BASED(SEM_ADDR),
             2 EXCL EVENT, 2 SHR EVENT,
             2 #_SHR FIXED BIN(15),
           (COMPLETION) BUILTIN,
           (ENQUEUE,DEQUEUE) ENTRY  OPTIONS(ASM INTER) EXT;
       CALL ENQUEUE;
       IF COMPLETION(SEM_ADDR->EXCL)~=COMPLETION(SEM_ADDR->SHR)
          THEN IF  COMPLETION(SEM_ADDR->EXCL)
                   THEN COMPLETION(SEM_ADDR->EXCL) = '1'B;
                   ELSE DO;  /* RELEASE OF SHARED RESOURCE */
                           SEM_ADDR-> #_SHR = SEM_ADDR-> #_SHR - 1;
                           IF SEM_ADDR-> #_SHR = 0
                              THEN COMPLETION(SEM_ADDR-> SHR) = '1'B;
                        END; /* OF ELSE DO */
       CALL DEQUEUE; RETURN;
END LIBRATS;
```

FIGURE 4. ROUTINES FOR SHARED ACCESS.

References

[1] IBM PL/1 CHECKOUT AND OPTIMIZING COMPILER LANGUAGE REFERENCE MANUAL, GC33-0009-3

[2] Tsichritzis,D., Bernstein, P ., OPERATING SYSTEMS, Academic Press 1974, pp 32-33

[3] Dijkstra,E.W., "Cooperating Sequential Processes", PROGRAMMING LANGUAGES, F.Genuys(ed.), Academic Press, 1968 pp.43-112

[4] Modell, H.S, Ward, R.G., "Language Additions To PL/1 For Controlling Concurrent Processes", Proceedings of the International Conference on Design and Implementation of Algorithmic Languages, JUNE 1976.

[5] MacEwen, G.H., A Programming Project For A Course In Operating Systems, Queen's University, Kingston, Canada

[6] Modell, H.S., Image Feature Extraction Using Parallel Processing, Master's thesis, UTA, 1976 64 pp.

[7] Presser,L., "Multiprogramming Coordination", ACM COMPUTING SURVEYS, Vol 7, No.1, March 1975, pp.21-44

[8] IBM OS/370 SUPERVISOR AND DATA MANAGEMENT SERVICES MANUAL, #GC28-6646

[9] McGowen,C., and Kelly,J., TOP-DOWN STRUCTURED PROGRAMMING TECHNIQUES, Petrocelli/Charter Publ., NY 1975, pp 164-203

ON THE TIME REQUIRED TO PARSE AN ARITHMETIC EXPRESSION FOR PARALLEL PROCESSING

Ross A. Towle
Federal and Special Systems Group
Burroughs Corporation
Paoli, Pa. 19301

Richard P. Brent
Computer Centre
The Australian National University
Canberra, Australia

Several algorithms that attempt to reduce the tree height of a parse tree of an arithmetic expression by using associativity and commutivity have been proposed [1] - [5]. Baer and Bovet [1] conjectured that their algorithm obtained a minimal-height tree. Later Beatty [2] proved this conjecture. Without distributivity, the upper bound on the tree height is: $1 + 2d + \lceil \log_2 n \rceil$ where n is the number of operands and d is the depth of parenthesis nesting [6]. The selective use of distribution reduces the upper bound on the tree height to $\lceil 4 \log (n-1) \rceil$ [7]. Several algorithms which use distribution have been proposed [7] - [9].

Using the algorithms of Beatty [2] and Brent [7] we can show that the use of associativity and commutivity adds $O(N)$ steps to the parsing process, while the use of associativity, commutivity, and distributivity adds $O(N \log_2 N)$ steps. Both algorithms work on parse trees produced by ordinary compilers and the times quoted are in addition to the ordinary parsing time.

We shall assume that an arithmetic expression contains N tokens (identifiers, constants, and operators). In calculating the number of operations required to parse an expression we shall count each instance of an arithmetic operation, logical operation, push on stack, pop stack, and store as one operation.

The results are summarized in the following theorem:

Theorem [10] Let E be any arithmetic expression with N tokens. The additional time to parse E is at most: 1. $13N + 8$ by Beatty's algorithm
2. $31N \log_2 N$ by Brent's algorithm

References

(1) J.L. Baer and D.P. Bovet, "Compilation of Arithmetic Expressions for Parallel Computations", Proc. IFIP Congress 1968, pp. 340-346.

(2) J.C. Beatty, "An Axiomatic Approach to Code Optimization for Expressions", Journal of the ACM (October, 1974), pp.613-640.

(3) J. Hellerman, "Parallel Processing of Algebraic Expressions", IEEE Trans. on Electronic Computers (January, 1966), pp. 82-91.

(4) J.S. Squire, "A Translation Algorithm for Multiple Processor Computers", Proc ACM 18th Nat. Conf., 1963.

(5) H.S. Stone, "One-pass Compilation of Arithmetic Expressions for a Parallel Processor", Comm. ACM (April, 1967), pp. 220-223.

(6) D. Kuck and Y. Muraoka, "Bounds on the Parallel Evaluation of Arithmetic Expressions Using Associativity and Commutivity", Acta Informatica (August, 1974), pp. 203-216.

(7) R.P. Brent, "The Parallel Evaluation of General Arithmetic Expressions", Journal of the ACM, (April, 1974), pp. 201-206.

(8) Y. Muraoka, Parallelism Exposure and Exploitation in Programs, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Report 424, (February, 1971), 236 pp.

(9) R.P. Brent, D.J. Kuck, and K.M. Maruyama, "The Parallel Evaluation of Arithmetic Expressions without Division", IEEE Trans. on Computers, (May, 1973), pp. 532-534.

(10) R.A. Towle, Control and Data Dependence for Program Transformations, Dept. of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-76-788, (March, 1976), 113 pp.

This work was sponsored by the National Science Foundation through grant US NSF DCR73-07980 A02 while the author was at the University of Illinois.

LAU SYSTEM SOFTWARE : A HIGH LEVEL DATA DRIVEN LANGUAGE FOR PARALLEL PROGRAMMING

by

O. Gelly, et al, Dpt of Computer Science, ONERA CERT, BP 4025 Toulouse, France

Summary

The LAU parallel system [1] is a contribution to the development of the software concept of single assignment [2] to parallel programming languages and parallel processor architecture [3].
This paper emphasizes some points among the main features of the high level language LAU :

Objects. A new attribute takes place in the object definition, and is defined as a set of environmental rules which tell HOW and WHEN the object may be manipulated by the program statements. The standard implicit value for this attribute is the "single assignment" rule, i.e. : any object may be assigned at most once during program execution. This rule leads to a parallel and deterministic execution of statements based on the "readiness" of their operands. For example, X=A+B placed anywhere in the program may be computed as soon as A and B have their (unique) values.

The user may also define "non standard" objects using a special command CREATE. The "environment" section characterizes the possible manipulations of the objects. The environmental expression makes use of the actual values of an input parameter sublist and operators that may be compared to Campbell's Path expressions. The object BUFFER defined below has its environment characterized by the couple (object operating on BUFFER, operation requested). WHO and WHAT will participate to the environmental expression, which means that :
- A WRITE operation, from P1, must occur first
- A sequence of one READ operation, from any one and one WRITE operation from P2, may follow.
- A COUNT operation from P3, if no more read requests at that time, will definitely close the manipulations of BUFFER.

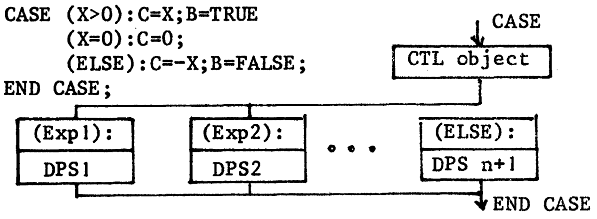
```
CREATE BUFFER (WHO,WHAT,QTY) (ANS) ;
IN : WHO,WHAT : EVENT ; QTY : INTEGER ;
OUT : ANS : INTEGER ;
LOCAL : BUFF, N : INTEGER ;
DO : CASE WHAT
        (WHAT = READ) : ANS = OLD BUFF ;
        (WHAT = WRITE) : BUFF = QTY ;
                         COUNT = OLD COUNT + 1 ;
        (WHAT = COUNT) : ANS = OLD COUNT ;
     END CASE ;
SYNC ON <WHO,WHAT>
     BY <P1,WRITE>.(<*,READ>.<P2,WRITE>)
        <P3,COUNT>
END CREATE ;
```

Statements. All statements must be considered as assignment statements. A statement is semantically defined by one or more Data Production Sets (DPS) which consist of a couple (set of statements S, set of objects to be computed, O).
Simple assignment statements are quite conventional, except that their semantics is purely directed by the data.

The EXPAND statement is simply an extension of vector operations, equivalent.

The CASE statement looks like a general CASE OF. A run time, one DPS will be activated, and will produce the object Outputs corresponding to the CASE statement.

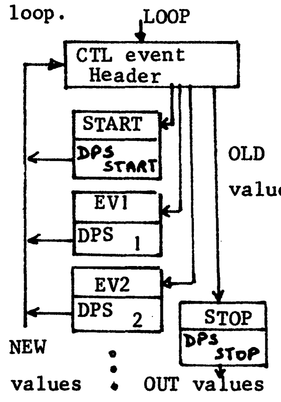
```
CASE (X>0):C=X;B=TRUE
     (X=0):C=0;
     (ELSE):C=-X;B=FALSE;
END CASE;
```



DPSi : S : statements nested in EXPi
O : all objects assigned between CASE and END CASE

If X=0, then CASE assigns C=0 B=NIL by DPS2

The loop statement has been designed to make iteration easy to devise in a parallel asynchronous environment. A loop header controls the value of a loop event, set initially to START and then at each iteration and activates the corresponding DPS. For any object in the OUT LOCAL section, a "local environment" is created by defining OLD X as the previous value of X, and NEW X or X as the value to be assigned at the current DPS activation. An implicit DPS STOP terminates the loop execution by assigning the actual outputs of the loop.



```
LOOP FIBN
  OUT:X,Y;
  LOCAL:N;
  (START):N=1;X=2;Y=1;
          FIBN=NEXT;
  (NEXT):CASE(OLDN=10):
           STOP LOOP FIBN
           (ELSE):
           X=OLDX+OLDY
           Y=OLDX;
           N=OLDN+1;
         END CASE
END LOOP FIBN;
```

As a conclusion, this language has been tested on about 50 problems and makes parallel programming easy to write and debug.
A compiler produces object code to the simulator (2).

References

[1] D.Comte,G.Durrieu,O.Gelly,A.Plas,J.C.Syre TEAU 1-9, Technical Reports,Authors'address.

[2] A.Plas, et al., LAU System architecture, These Proceedings.

[3] Tesler L.G.,and Enea,H.J. A language design for concurrent processes, Proc AFIPS SJCC68 p. 403-408.

A HIGH LEVEL LANGUAGE ORIENTED MULTIPROCESSOR[+]

Mario F. de la Guardia and James A. Field
Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, Canada

Abstract -- A stack organized multi-processor whose instruction set is oriented to high level language processing is described. The proposed areas of application are communications control and industrial process control. The memory is organized on a segment basis so that procedure calls, parameter passing, and process suspension and activation have very low overheads. Dijkstra semaphores are used to co-ordinate all signalling activities between processes. All processing of semaphores, ready lists, etc., is performed by hardware (firmware) allowing the user to program the machine in a high level language.

Introduction

There are many computer controlled systems in which the control activity logically consists of several co-operating parallel operations. Examples are industrial process control, and communications control activities such as message switching and front end processors for teleprocessing. In such systems one can identify processes that could be performed in parallel, with little interaction, on the computer. However, until recently the only computers available (with suitable cost to performance figures) for such applications were the standard single processor minicomputers. In such processors no advantage could be gained from the parallel nature of the problem, and indeed, the frequent necessity of producing pseudo-parallel service to meet the real time requirements placed a significant execution time and/or memory space overhead on the system. Now, however, the general availability of microprocessors has made it feasible to consider using a set of co-operating microprocessors to perform the desired function. In some situations the power of an individual microprocessor will be well matched to the requirements of an individual process. In such cases a set of loosely coupled microprocessors will conveniently and effectively meet the total system requirements. In other cases such divisions are not possible and a more tightly coupled and more interactive configuration of microprocessors is required. Several authors have discussed such possible configurations of mini and microprocessors with the goal of greatly improved service to cost ratios compared to single processor systems [1-4]. Unfortunately, the development and general applications of such systems suffers from a serious handicap: neither the basic hardware nor software (language) really "thinks" parallel.

[+]This work was supported by the National Research Council of Canada.

It was desired to develop a system, probably microprocessor based, that could exploit current LSI technology for systems of the above parallel nature. The approach was to consider the current software and hardware limitations to such parallel activity, and to develop a structure that did not contain these restrictions.

Consider first the software problems: these are language and operating system. Most mini and microprocessor systems are programmed at the assembly language level. This is frequently justified on the basis of program efficiency. However, given the current state of compiler development there is no reason why efficient code cannot be generated by an appropriate high level language (HLL) compiler for any processor in question. True it may require support by a larger computer, and for highly optimized code may be slow, but this is easily justified by the higher portability, reliability and maintainability of the resulting program. This is particularly true where the program will be used for a considerable period of time after being developed. It might be noted that such HLL programming is being very successfully used in the SL-1 PBX telephone switching computer [5].

What form should the HLL take? A safe choice would seem to be a block structured language along the Algol model. Such languages have been widely used and studied. Consequently, a good understanding exists of what data structures are required, what hardware features are needed, and very importantly, how compilers should be written. These languages also support recursion, which as well as being a useful feature, requires many of the mechanisms that are required to support parallel process execution, and hence, such languages are logically and conveniently extendable to parallel concepts. Current planning is to use a modification of the Algol-W language [6]. (One modification would be the deletion of the goto, thus not only forcing more structured coding, but also resulting in a more efficient compiler [7].)

With respect to operating system software, there should be none. Features such as schedulers, loaders, I/O handlers, etc., are software patches between the user language and the hardware. With the availability of firmware support they could, and should, be made transparent to the user.

Turning now to a consideration of the hardware, the most obvious feature is that the hardware should support the programming language. When using a HLL with conventional machines the operations of procedure call, actual parameter passing, and process suspension and restart have high overheads. If the processor has machine language instructions specifically related to these, and other complex HLL constructions, more efficient operation will obviously result. Further, by closely matching the hardware to the language, the operations of the compiler closely approach those of a traditional assembler; there is a machine instruction for every major language construction. (However, the final step to direct execution of the HLL is not being considered at this time. HLL has considerable symbolic inefficiency compared to a well designed machine code, e.g., a HLL program occupies at least the same amount of program store, and requires more decoding operations, than HLL oriented machine code. Thus more program store, and either more decode hardware or slower execution would result.)

For the general capabilities outlined above for the hardware, it appears that provision of the services by firmware may be the best solution. Current planning is that such firmware may be supplied by microprocessors with read only memory. However, the actual mechanism is not significant, and in the following sections no distinction is drawn between firmware and hardware.

The general structure of the proposed machine is shown in Figure 1. It allows several processor elements but will operate with only one with no system changes. A key component is the ready list and priority structure. This is the mechanism that allows parallel execution of processes. It services both input/output and software interrupts, and maintains the list of processes awaiting processor availability. This is done via the Dijkstra P and V operations [8]. (The same techniques are used to share mutual data areas.) It is accessible by all processors, but with appropriate locks to prevent simultaneous access. It will be discussed more fully later.

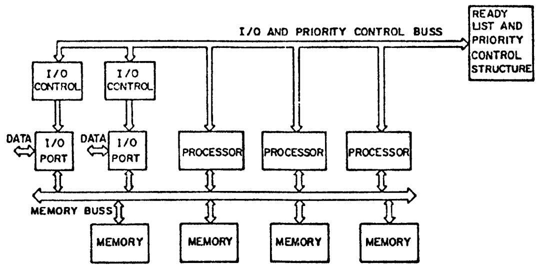


Figure 1. Computer structure.

Input/output devices are seen as consisting of two components, the controller and the memory port. For simple devices (one word at a time), the controller is sufficient. For high speed devices the controller is associated with a memory port, and it co-ordinates the flow of information through that port. Interrupts generated via the I/O controllers will be sent to the lowest priority processor via the ready list and priority unit.

The memory is considered to consist of two sections: the program store and the data store. This distinction is made because the program store contents are fixed by the compiler, while the data store contents are dynamic. Consequently it is convenient to use different addressing mechanisms, and hence, logically, if not physically, separate memories. The instructions are variable length, consisting of 8 bit elements, and are fetched 16 bits at a time. This operation is quite conventional and will not be considered further. The significant feature is the data memory organization. The basic approach is to use a stack organization. However, it is extensively modified to give good performance in the high overhead areas of procedure call and process suspension and restart. This will be considered in detail in the next two sections. It should be noted that several of the concepts discussed are used in the Burroughs computers [9]. However, the Burroughs machines are large scale systems. Here the emphasis is on small machines with significant real time parallel activity and frequent process suspension and restart.

Stack Organization of Data Blocks

The concept of using a stack oriented data structure follows from the choice of a block structured language since such languages release data blocks in the reverse order to creation, i.e., a first in last out stack. There are actually four distinct operations in the system that may be conveniently organized around a stack structure. These are (1) allocating space for data blocks, (2) recording return environments for procedure calls, (3) evaluating arithmetic expressions, and (4) the passing of parameters to called procedures. Neglecting for the moment the question of multiprocess operation, we will consider how these four services may be combined in a single stack discipline.

Experience with compilers for block structure languages indicates that (1) and (2) above can be conveniently combined into a single stack. The first items on the stack are the saved environment parameters, followed by the data area for the block or procedure. Similarly, investigations of direct HLL execution, or HLL oriented machines have used such an organization [5,9,10,11]. All these implementations consider the address of a variable to consist of a number pair: one number selects the data block via a display and the other locates the item within the block.

As is well known, a stack may be used for the efficient evaluation of arithmetic expressions in the machine code equivalent of Polish postfix notation. This form of code is not only simple for compilers to generate, since it avoids the problems of register assignment and intermediate result storage associated with register oriented computers, but is also a compact code since only a few instructions require memory addresses. The simpler compiler is not insignificant in view of the earlier remarks regarding the desire for an assembler-like compiler. That the arithmetic evaluation stack can be conveniently combined with the data block stack has been demonstrated [5,9,11]. The disadvantage of using the same stack for expression evaluation is speed. If a separate stack were used it could be composed of higher speed registers than the normal memory cells. Thus having only one stack apparently limits the arithmetic operations to memory speed. However, this can be avoided, if necessary, by providing a few high speed registers to serve as the top few stack locations. This would allow most arithmetic expressions to be evaluated without using (slow) memory for stack area. When it is necessary to advance the stack pointer to create a new data block, the content of these registers would be automatically stored in memory. If the registers are only used during arithmetic operations they would seldom be forced into memory. Further, if once stored into memory they are never reloaded, such register saving puts very little overhead in the system operation. The key item is to prevent repeat loading/storing of these registers in memory and there must, consequently, be no attempt to keep them full.

The same stack can also be used for passing parameters during a procedure call. This is done by leaving a "hole" in the stack (by incrementing the stack pointer) for later recording of the calling environment, and then placing the parameters on the stack. (This is especially convenient if the actual parameters are expressions since they must be evaluated on the stack.) The procedure called then considers this section of the stack as the initial part of its data block. One point must be noted; the compiler must do complete checking as to the number and type of arguments, and their method of being passed. (This is not an unreasonable request to make of a compiler in any system.) The methods of passing parameters being considered are value, result and array. Result does not place items on the stack; rather it copies items off the stack after the return from the procedure call. Call-by-name could be implemented but does not appear to be of value for the applications being considered.

As an example consider the program fragments shown in Figure 2. The procedure demo has three formal parameters and four local variables. In the calling sequence the machine instruction space advances the stack pointer by four words leaving space for the later recording of the calling environment (see Figure 3). The current values of the two value formal parameters u and v are then generated on the stack. The machine then executes the instruction pbegin 3,2,11 (which corresponds to the procedure declaration and the enclosed integer type declaration). Here the machine uses the non-result formal parameter count (i.e., 2) to locate the environment save area address relative to the stack pointer and record the appropriate values, and uses the block size (i.e., 11) to set the stack pointer to the next free location. The body of the procedure is then executed. When the procedure is complete the result formal parameter w is placed on the stack. The machine code pend then releases the data block and transfers the specified number (1 in this case) of result parameters from the top-of-stack before block deletion to the top-of-stack after block deletion(a). The pop machine code in the calling block then stores the value in the actual call-by-result parameter.

In the implementation of block structured languages on conventional computers the mechanisms of acquiring/releasing data blocks, and the transferring of parameters to procedures, have been high overhead areas. It can be seen that the above approach has placed the overhead operations in the hardware where it can be performed more efficiently than in conventional software.

Another area of potential high overhead is the maintaining of a valid display. The display must be updated on every block or procedure entry or exit. In computers with insufficient base registers to maintain the display, this usually requires that a new copy of the display be generated for every new data block created [12]. This is partially due to exiting problems created by uncontrolled use of goto. In the proposed system goto is not permitted and there are sufficient base registers for the display. Under these conditions the only slow operation is procedure exit. Consider first the block entry

```
          HLL program                 machine code

procedure demo
        (integer value u,v;
         integer result w);

   begin integer p,q,r,s;         demo pbegin 3,2,11
   .                                   .
   .                                   .
   .                                   .
   end demo;                           push w
.                                      pend 1
.                                      .
demo(a,b*c,d);                         space
.                                      push a
.                                      push b
                                       push c
                                       mult
                                       call demo
                                       pop d
```

Figure 2. Example of actual and formal parameter association.

(a) This transfer will be quite rapid if the stack is augmented with high speed registers. In that case no data movement is needed; rather the various stack pointers are modified.

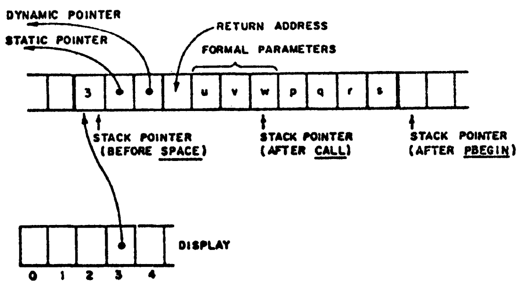

DYNAMIC POINTER
STATIC POINTER
RETURN ADDRESS
FORMAL PARAMETERS
3
u
v
w
p
q
r
s
STACK POINTER (BEFORE SPACE)
STACK POINTER (AFTER CALL)
STACK POINTER (AFTER PBEGIN)
DISPLAY
0
1
2
3
4

Figure 3. Data block structure for the procedure demo.

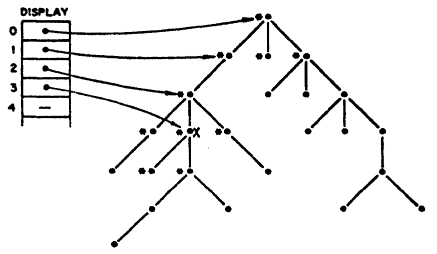


Figure 4. Enclosure tree indicating procedures accessible to block x.

and exit operations. When a block at level n is entered control must come from level n-1. Thus the display is correct up to level n-1, and when the level n data block is created, all that must be done to the display is put the address of the new block in the nth display register. At block exit no action is needed since the display is already valid for level n-1. With respect to procedure calls, consider the enclosure tree shown in Figure 4. Suppose a call is being made from block x to some procedure. The procedures that may be referenced are marked with an asterisk. Now, at each of those locations the display is correct up to the previous level, and only the address of the new data block must be inserted. However, when the called procedure terminates, varying amounts of the display must be regenerated to restore the environment of block x. This restoring is done by tracing back the static pointers in the environment area of the data blocks. The number of levels to be restored is given by (level of block x - level of called procedure + 1). In normal circumstances this is quite small and will be done rapidly. One memory reference must be made to obtain each value of the display to be restored.

Multiprocess Stack Organization

The above approach is quite satisfactory for a single process system. However, when there are several parallel processes, each must have its own stack since each process will be dynamically acquiring and releasing data blocks. In conventional systems this is handled by using a memory allocation routine, and a pure stack allocation technique does not result. Rather a process explicitly requests space for a new block from the allocator routine, and then explicitly releases the block to the allocator routine on block exit. This means the stack for a given process may be fragmented if other processes make intervening requests. This creates difficulties in the use of the stack for argument passing and expression evaluation due to the need for mechanisms to "bridge" over the "gaps" created by the non-sequential allocation. The system also requires the services of a "garbage collection" routine to reassemble small fragments of memory space into usable blocks. As the extensive literature on this latter item indicates, it is a non-trivial task when using conventional software methods. Generally speaking, software memory allocation schemes add considerable overhead time to program execution.

Another troublesome feature of multi-process operation is that a process may be suspended and then re-activated. During the idle period the processor will be assigned to some other process. This requires that the display registers be saved when the process is suspended, and restored when it is re-activated. Since in many applications the running time of a process between activation and suspension may be quite short, the saving and restoring of registers could contribute significant overhead. To reduce the overheads, hardware solutions to the memory allocation and process suspension/restart problems were developed. The first step is using a memory segmenting scheme under which each parallel process has an effective memory space extending from word zero to its stack limit value. When the stack pointer is advanced past the stack limit the hardware automatically assigns a new memory segment to the process and adjusts the stack limit.

Figure 5 shows the address decoding scheme that provides the desired operation. It is proposed that the data memory shall consist of up to 256 segments of 1024 sixteen bit words[b]. Each process may have a stack size up to 64 segments long. Associated with each running process are 64 high speed registers (segment registers) that hold the true memory address of the segments that constitute the process' stack. Figure 5 shows both the block level address (via the display) and the segment address decoding of a memory reference instruction yielding the 18 bit actual memory address. Other addresses, such as the stack pointer, bypass the block level decoding.

[b]The number and size of segments is not critical to the method.

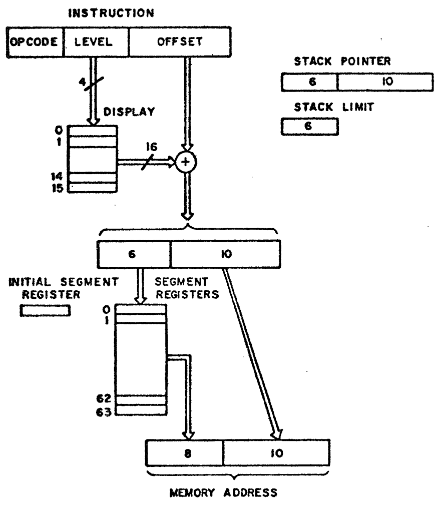


Figure 5. Memory address decoding system.

As indicated earlier, incrementing the stack pointer past the current stack limit causes the hardware to acquire a new segment from the free segment pool, place its address in the appropriate segment register, and adjust the stack limit. Similarly, when the stack pointer drops below a segment boundary it would be possible to release the segment to the free segment pool and adjust the stack limit downwards. It is possible, however, that the process would then shortly advance the stack pointer across the boundary thus creating a request for a segment. To eliminate this overhead, it is proposed that the release of vacated segments does not occur until a process is suspended, and at that time all full segments above the stack pointer be released and the stack limit adjusted accordingly.

The memory segmenting solves the problem of multiple stacks and "garbage collection," but it has apparently compounded the problem of saving/restoring the display registers on process suspension/restart by adding the segment registers which must be treated in the same fashion as the display registers. However, the presence of segments allows the permanent assignment of save areas for these registers. The segment and display registers will be saved in words 0 through 63 and 64 through 79 respectively of the segment pointed to by the initial segment register. Now, since a program tends to exhibit "locality" of data reference, and further, since it is anticipated that many processes will be active for only short periods, it follows that most display and segment registers will not be required during any given process activation. Therefore it is proposed that on process reactivation only the initial segment register is reloaded. All the segments and display registers will have a flag set to indicate they are not initialized. If during code execution reference is made to an uninitialized register a correct copy is fetched from the appropriate word of the initial segment and the register is marked as initialized. In this way only items that are actually needed will be restored. With regard to saving the registers, a similar technique could be used to mark those that have been modified, and then when the process is next suspended the marked registers would be saved. This introduces a variable delay in the suspension process which may be undesirable if the suspension is caused by higher priority pre-emption. An alternate method is to record in the initial segment save area any modification made to a segment or display register at the time the modification is made. Then none of the registers must be saved when the process is suspended. This point needs further study to select the optimum method.

Co-ordinating Parallel Activity

As indicated earlier the system allows signalling, and mutual resource protection, by means of "semaphore" operations based upon the P and V operations of Dijkstra. The semaphore variables use two words each. The first is the semaphore value, and the second is a link to the first process waiting (blocked) on that semaphore. Figure 6 shows a semaphore with three blocked processes. The second word also indicates the priority of the blocked process. The actual link value is the initial segment address of the blocked process. In the initial segment word 80 is used to continue the chain to the next blocked process, etc. Words 81 and 82 are used to save the stack pointer and restart address respectively of the blocked process.

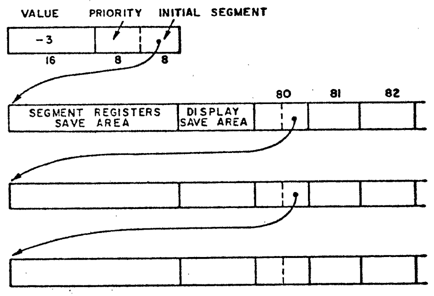


Figure 6. Semaphore and blocked processes.

When a V operation is performed on a semaphore the first process on its queue is transferred to a ready list. There will be a separate ready list for each desired priority level with the head of the list in a hardware register. Figure 7 shows a typical ready list. When a processor becomes idle, it selects the top item on the highest priority non-empty ready list. When a process is released (V operation) whose priority is higher than one or more of the running processes, and no processor is idle, an interrupt will be generated in the processor running the lowest priority process. The only function of the interrupt is to cause the processor to suspend the current process, and return it to the top of the appropriate ready list. The processor is then idle, and will automatically select the highest priority process from the ready lists. Obviously, to prevent confusion there must be an arbitration mechanism so that only one processor at a time can manipulate semaphores and the ready lists.

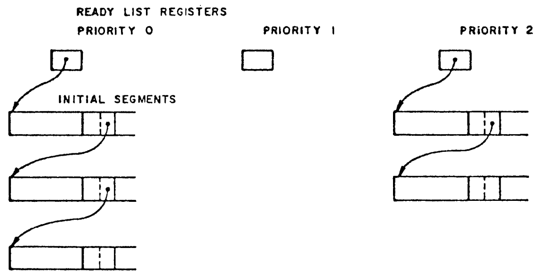


Figure 7. Ready list structure.

Input/output interrupts will be handled by recording the desired priority and the address of a semaphore (initial segment number plus 16 bit address) in the I/O device controller's registers. When an I/O device wishes to cause an interrupt, it generates a signal similar to the higher priority process ready interrupt. This signal causes the lowest priority processor to read the semaphore address from the input/output device and perform a V operation upon it. It is not necessary for the processor to suspend the running process unless the V operation releases a higher priority process.

To allow inter-process communications it is necessary that parallel processes can share common data. This will be achieved by a running parent process spawning a child process with access to the parent's data blocks. This is done by acquiring an initial segment for the child process and initializing its segment and display register save area with a copy of the appropriate parts of the parent process' save area. For example, suppose the child process was at block level 4 and hence had access up to block level 3 of the parent, and that this occupied (due to recursion or large arrays) from segment 0 to part way up segment 5, then segment registers 0-5 and display registers 0-3 of the child would be copies of the parent. The next available segment (6 in the example) is the initial segment of the child process. The start address of the child is stored in location 82, and the value of the initial stack pointer address (word 83 of segment 6 in the example) is stored in word 81 of the initial segment. The initial segment address is then added to the ready list of the desired priority and the child process is established. To prevent nasty accidents due to the parent process releasing its segments while the child is still using them, the compiler must insert semaphores that prevent such release until the child process has terminated.

Concluding Remarks

In the above the emphasis has been on the data store management used to minimize overhead during environment changes. The constructions adopted do not in any way inhibit flexible data manipulation. For example, it is expected that the Algol-W record class concept could be easily implemented in addition to arrays and simple variables. It might also be noted that the segmenting features could be extended to program store if a dynamic programming environment were needed. Further, it would be possible to extend the segmenting concepts to allow paging activity to/from a secondary store if desired.

At the moment the design is nearing completion on two versions of the processor: a microprocessor based system and a microcoded logic design. These will be evaluated with respect to expected performance and cost.

Reference

[1] A.C.M. Chen and W.D. Barber, "Multi-Microprocessor System for Industrial Control" Summary, Proc. 1975 Sagamore Computer Conference on Parallel Processing, August 1975, p. 105.

[2] A. Baum and D. Senzig, "Hardware Considerations in a Microcomputer Multiprocessing System," Digest of Papers, Tenth IEEE Computer Society International Conference, February 1975, pp. 27-30.

[3] D. Schutzer, "A Modular Approach to the Design of a Communications Control Processing Center - Pros and Cons," Digest of Papers, Eleventh IEEE Computer Society International Conference, September 1975, pp. 31-33.

[4] M. Irland and E. Manning, "Multiprocessor Simulation Using Minicomputers of Packet-Switched Data Networks," Proc. A.I.M. International Meeting on Mini-Computers and Data Communications, Liege, January 1975.

[5] B. Kelsh and P. Lewis, "Key to System Features: SL-1 Software," Telesis, Vol. 4, No. 3, Fall 1975, pp. 91-95.

[6] N. Wirth and C.A.R. Hoare, "A Contribution to the Development of ALGOL," Comm. ACM 9,6, June 1966, pp. 413-431.

[7] A.W. Wulf, "Programming without the GOTO," Information Processing 71, North-Holland, 1972, pp. 408-413.

[8] E.W. Dijkstra, "Cooperating Sequential Processes," in "Programming Languages" (F. Genuys, Ed.), Academic Press, New York and London, 1968.

[9] E.I. Organick, "Computer Systems Organization: The B5700/B6700 Series," Academic Press, New York and London, 1973, 132 pp.

[10] L.S. Haynes, "Structure of a Polish String Language for an Algol 60 Language Processor," ACM-IEEE Symposium on High-Level-Language Computer Architecture, University of Maryland, November 1973, pp. 131-137.

[11] M.J. Lutz, "The Design and Implementation of a Small Scale Stack Processor System," Proc. National Computer Conference, 1973, pp. 545-553.

[12] D. Gries, "Compiler Construction for Digital Computers," Wiley, 1971, p. 195.

THE IMPACT OF APPLICATIVE PROGRAMMING ON MULTIPROCESSING*

Daniel P. Friedman
David S. Wise
Computer Science Department
Indiana University
Bloomington, Indiana 47401

Abstract -- Early results of a project on compiling stylized recursion into stackless iterative code are reviewed as they apply to a target environment with multiprocessing. Parallelism is possible in executing the compiled image of argument evaluation (collateral argument evaluation of Algol 68), of data structure construction when suspensions are used, and of functional combination. The last facility provides general, concise expression for all operations performed in LISP by mapping functions and in APL by typed operators; there are other uses as well.

Keywords: Functional combination, suspensions, recursion, parallelism, compiling, LISP.

CR categories: 4.32, 4.29, 4.12, 4.13.

Introduction

The purpose of this paper is to review the implications of recent results in recursive programming under a highly parallel execution environment. These are early results of a project aimed at the compilation of stylized purely recursive code. They have been presented elsewhere [5], but the implications of this type of compilation for highly parallel target code have not been gathered in one paper.

As programming tools these results appear as enhancements to applicative programming, enhancements we find necessary to strengthen classic (LISP [20], ISWIM [3] & [19]) recursive languages to express preimages of classic iterative programming techniques. While iterative programming is better developed, more familiar, and better understood than applicative programming, we strongly believe that it is unsuited to modern programming problems. Iterative programming has its roots in Turing's theoretical work. It grew with the first computers and matured through the development of programming languages (FORTRAN and descendants) which at first attempted to model iterative machine architecture and later, because of their universal acceptance, proceeded to determine that architecture. The work of Gödel and Church, contemporary with Turing's, supports another philosophy of programming which we feel is required to conceptualize solutions to problems for implementation on modern hardware.

We adopt a philosophy requiring all programs to be expressed as functions. There are no explicit loops (hence no goto controversy), no assignment statements (only parameter bindings), and no explicit input/output functions (instead input files are taken as arguments to the main program and output files are results [9]). The language described below has been implemented semantically in a single processor environment [10]. The techniques described here do not change the semantics of the language as far as computed results are concerned. They will, however, alter a program by allowing concurrent processors to alter the space requirements as necessary to allow computation to proceed.

An issue not discussed here but implicit in all our designs is the style in which the programmer is expected to express his algorithm. Stylized recursion [5] is a methodology of formulating recursive programs which encourages good, efficient program structure and permits effective analysis and transformation before the code is executed. It is during this compilation phase that we expect that parallel processing can be specified. The programmer does not concern himself with the possibilities and pitfalls of parallelisms; the compiler selects the parallelisms from his stylized code and provides the synchronization of the processes it has identified. Our control structures allow more of this automatic parallelism selection than classical iterative control structures [16].

The remainder of this paper is in five parts. Only the last explicitly discusses parallelism; the first four develop a language with trivial syntactic structures but with semantics which have only been recently proposed and which allow a remarkable degree of parallelism in interpreting applicative languages. The first section introduces the elementary syntax of the language whose only control structure is a function call; an obvious parallelism allowed is collateral argument evaluation. The second feature introduced is functional combination, whereby conceptually parallel

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

applications of several functions may be dispatched across multiple arguments yielding multiple results. Third, an extension of functional combination to arbitrary instances of the same function or the same argument allows a simple representation for the concept of "mapping" or "pipelined" operations on homogeneous structures. The fourth feature, provided by suspended argument evaluation in the primitive <u>cons</u>tructor function, allows for massive unstructured parallelism in a system with thousands of processors. The last section develops possible interpretations of these features at run-time; the reader more familiar with parallelism than with applicative programming might scan it first in order to cast his interpretation of the four language sections in terms of something more familiar.

<u>The Language</u>

The only structure in the language is a parenthesized acyclic list. The programmer may use it to construct arrays (e.g. a list of lists), trees, and directed ordered acyclic graphs (doags). (n.b. This does not mean that the run-time structures are necessarily linear or acyclic -- the compiler may have changed them.) Functions that manipulate these data may be built from a given set of elementary list operations.

Lists are composed of elementary items or other lists. An elementary item is either an identifier (which may be bound to another value) or an integer (which is implicitly bound to itself). For example, the five following structures are legitimate as data:

```
123
FRED
(2 3 4 5 6)
()
(FRED (8) (2 MANY () (GREEN)) BANANAS)   .
```

A program is a function which takes as data a list of the above sort and generates a list or an elementary item as a value. The program, however, never uses the parenthesis notation explicitly.

The first programming notation is square brackets: a bracketed sequence evaluates to the list of the evaluated items of the sequence in order. For example, [6 5 4 3] evaluates to (6 5 4 3). Let x have the value (2 4 6 8) and let y have the value (B A N A N A S) . Then [x y] evaluates to
  ((2 4 6 8)(B A N A N A S)) .
Bracketed sequences provide only for creating lists of fixed size and therefore they can be associated with record structures of other languages. There is also a list building function, <u>cons</u>, for building lists of undetermined length; but before introducing it we must introduce the syntax for function invocation.

Function invocations are represented by a pair of items enclosed by angle brackets: <f ℓ> . The function position, here denoted by f, indicates the operation to be performed upon the argument list ℓ. Combined with square brackets this functional syntax is very suggestive of standard mathematical notation. Instead of min(i,j) we write <min [i j]> , and <sum [2 3 4 5 6]> evaluates to 20. (See also [1] and [13] for similar applicative expressions.) With the binding of x from above, <sum x> evaluates to 20; this case illustrates that the argument list need not be explicitly bracketed although it usually is.

A most important primitive is <u>cons</u>; it takes two arguments, an item and a list, and returns the list whose first element is that item and whose remainder is the original list. Thus <cons[2 y]> evaluates to (2 B A N A N A S) . Two complementary operations, <u>first</u> and <u>rest</u>, return the first item on a list and the list without the first item, respectively. <first[x]> evaluates to 2 and <rest[y]> evaluates to (A N A N A S) . The semantics of these three functions are particularly interesting [8], and we shall return to them in the next section.

We shall use other elementary functions without definition; their meaning is obvious from context. These are often arithmetic, like <u>sum</u>, and include simple predicates: <u>null</u> tests whether its argument is an empty list, and <u>zero</u> tests if its argument is 0. Example functions are presented by relating a prototype invocation to its definition in terms of a conditional expression. This definition is presented as an alternating sequence of tests and values whose interpretation is assisted by the insertion of the "commenting words" <u>if</u>, <u>then</u>, <u>elseif</u>, and <u>else</u>. For example,

```
<min[i j]> ≡
  if <less[i j]> then i
  else j
```

can be abbreviated by

```
<min[i j]> ≡
  <less[i j]>  i
    j .
```

The tests are evaluated in sequence until one succeeds; the value immediately following that test is the value of the function. If no test succeeds then the value of the function is the value of the last expression in the sequence if the sequence is of odd length (the <u>else</u> part), or rarely the empty list if the sequence is of even length.

As an example we present the definition of the function allrember which removes all members equal to its first argument from the list which is its second argument.

```
<allrember[e ℓ]> ≡
  if <null[ℓ]> then []
  elseif <same[<first[ℓ]> e]>
        then <allrember[e <rest[ℓ]> ]>
  else <cons[<first[ℓ]>
             <allrember[e <rest[ℓ]> ]> ]> .
```

It is also possible to define functions which take an arbitrary number of arguments in the same manner. An example is the function concat which returns a list which is the concatenation of all its arguments (each of which is a list). An auxiliary function, append, is required which concatenates just two lists.

```
<concat ℓs> ≡
  if <null[ℓs]> then []
  elseif <null[<rest[ℓs]>]>
        then <first[ℓs]>
  else <append[<first[ℓs]>
               <concat <rest[ℓs]>> ]> ;
<append[ℓa ℓb]> ≡
  if <null[ℓa]> then ℓb
  else <cons[<first[ℓa]>
             <append[<rest[ℓa]> ℓb]> ]> .
```

Integers may be used as functions; as a function the integer i simply returns its i^{th} argument. One use of this notation provides for array subscripting: if c is bound to a list of lists (a matrix) then <3<5 c>> evaluates to the third item in the fifth list (or the entry in the third column of the fifth row). The integer 1 may also be used as an identity function, often with the "invisible argument marker" symbol #.

The symbol # evaluates to a token which is ignored as a parameter to a function. Its evaluation is therefore useless except as an eventual argument to some function; in that role it acts much like the numeral zero: as a place-holder in argument structures with no ultimate meaning itself. For example, if d is bound to the evaluation of

[# # 9 # 15 # #]

then <1 d> evaluates to 9, <2 d> evaluates to 15, and <3 d> diverges since there is no third item in d taken as a parameter list. A list like d is often used in conjunction with functional combination (below).

Functional Combination

Functional combination is described elsewhere in some detail [6] and [7]. It provides the framework which allows one recurrence to accumulate several results in the same way that a single iterative traversal of data may yield several summary statistics. We describe its syntax and semantics formally here. The hallmark of functional combination is the occurrence of a list in the function position. In first order languages (where forms cannot evaluate to functions) this can only happen if an explicit list (within brackets) appears where a function is expected:

$$<[f_1\ f_2\ \ldots\ f_{m_0}]\ [\rho_1\ \rho_2\ \ldots\ \rho_n]> .$$

The list immediately following the left angle bracket is called a combinator and is not evaluated. Instead each f_j is presumed to be a legitimate function; either it has a definition as a function or it too is a combinator. Any f_j must require at most n arguments; its arguments are extracted from the structure of the arguments to the combinator, ρ_i, each one is presumed to be a list.

The semantics of functional combination depends on the lengths of the arguments and of the combinator itself. Let m_i be the length of ρ_i, the i^{th} row. Let $m = \min_{0 \le i \le n} m_i$.

The result of evaluating the form with a combinator as its function is a list of length m. The j^{th} element in that list is the result of

$$<f_j\ [<j'\ \rho_1>\ <j'\ \rho_2>\ \ldots\ <j'\ \rho_n>]\ > .$$

(The integer function j' is the same as the function j except that a token evaluation of # is counted in selecting the result. If the result of applying j' is an instance of # it is passed as a parameter to f_j, which ignores it.)

In full blown form we have

$$\begin{aligned} <[f_1\ f_2\ \ldots\ f_{m_0}]\ [\rho_1\ \rho_2\ \ldots\ \rho_n]> = \\ [<f_1\ [<1'\ \rho_1>\ <1'\ \rho_2>\ \ldots\ <1'\ \rho_n>]\ > \\ <f_2\ [<2'\ \rho_1>\ <2'\ \rho_2>\ \ldots\ <2'\ \rho_n>]\ > \\ \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\ <f_m\ [<m'\ \rho_1>\ <m'\ \rho_2>\ \ldots\ <m'\ \rho_n>]\ >]. \end{aligned}$$

An elegant interpretation of the evaluation of such a form arises from viewing the result of evaluating each ρ_i as the i^{th} row of a matrix whose columns are then referred to as γ_j for $1 \le j \le m$. The result of evaluating the entire form is that of

$$[\langle f_1\ \gamma_1\rangle\ \langle f_2\ \gamma_2\rangle\ \dots\ \langle f_m\ \gamma_m\rangle] \quad .$$

Thus the evaluation procedure can be described as an evaluation of arguments in row-major order with parameters passed to functions in column-major order. The derivation of m as a minimum implies a "guillotine rule" which causes a "jagged" matrix of arguments to be truncated at the narrowest width. In the case that m = 0 the result of the evaluation is the empty list. As an immediate result the list [] is defined as a constant function: <[] ℓ> evaluates to the empty list regardless of the binding of ℓ.

In order to facilitate the matrix interpretation of functional combination its invocation will appear with the arguments on separate lines and vertically aligned to suggest the columnar relationship. Furthermore, names of functions which return results of fixed length will be hyphenated to suggest the meaning of each component of the answer. For example:

```
<[sum product quotient difference] [
  [ 0     1       63        19     ]
  [ 1     3       #         13     ]
  [ 0     3       9         #      ] ]>
```

evaluates to (1 9 7 6).

A more interesting example illustrates the power of functional combination as related to recursive programming. The function <u>ℓt-eq-gt</u> takes a list of numbers and a numeric value as parameters and returns three results corresponding to the three components of the partition of the list by that value: those less than, those equal to, and those greater than it. The construction of the partition is accomplished by a single linear recursion over the list. Since operations like this are common in programming (for example, it is the key step in the Quicksort Algorithm [14]), it is important that they be expressible in a form analogous to the simple loop available to iterative programmers.

The following example uses functional combination three times in essentially the same way: the pattern of invocation is <[...] [...]> which suits the row/column description given before. An invocation may also appear as <[...] ℓ> where ℓ is bound to a matrix which will be decomposed to extract parameters in the manner described above. It is also possible to write something of the form <[...] <...>> which indicates that the matrix will be the result of invoking a second function.

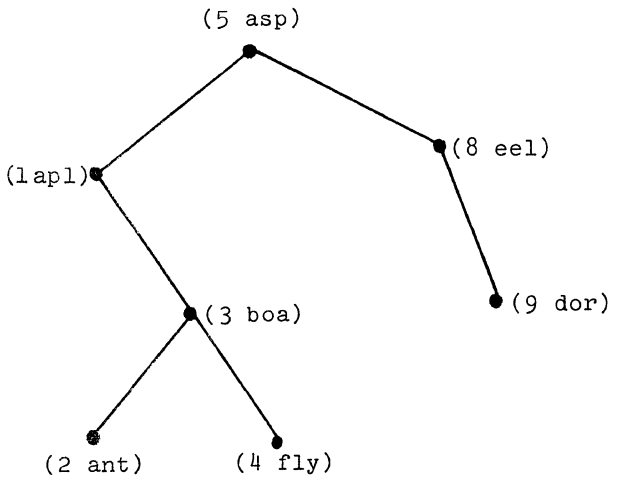
```
<ℓt-eq-gt[ℓ v]> ≡
  if <null[ℓ]> then [ [] [] [] ]
  elseif <less[ <first[ℓ]> v]>
        then <[cons        l l][
              [<first[ℓ]> # #]
              <ℓt-eq-gt[ <rest[ℓ]> v]> ]>
  elseif <greater[ <first[ℓ]>v]>
        then <[l l cons          ][
              [# # <first[ℓ]> ]
              <ℓt-eq-gt[ <rest[ℓ]> v]> ]>
  else <[l cons       l][
        [# <first[ℓ]> #]
        <ℓt-eq-gt[ <rest[ℓ]> v]> ]> .
```

Another application of functional combination involves the invocation of the function being recursively defined with the combinator. We present an example in which the defined function appears twice, resulting in two recursive invocations. In a deep recursion the invocation pattern generates a binary tree: at the $n^{\underline{th}}$ level the results are determined by the results of 2^n functional combinations which dispatch 2^{n+1} recursive calls. That tree structure is no accident since the example is concerned with searching binary trees [17] (those whose inorder [18] traversal visits the nodes in order of their keys). Let ℓ be an unsorted list of perhaps duplicated keys. We present a function, <u>quickbatch</u>, which probes <u>tree</u> to extract any information for every key in ℓ and returns a list of the associations for those keys which had information planted in <u>tree</u>. The list will be returned in ascending order of keys; and the search will be batched [21] , so that every subtree is visited at most once.

Define a binary tree to be () or a list of three items: (left information right). <u>Information</u> represents the data stored at the root of the tree whose subtrees are <u>left</u> and <u>right</u>, respectively. In this case <u>information</u> is an association of a <u>key</u> and data. The invocation <key[tree]> extracts the key from the root of the non-null tree; the definition requires that this key be greater than every key in the <u>left</u> subtree and less than any in the <u>right</u> subtree.

```
<quickbatch[ℓ tree]> ≡
  if <null[ℓ]> then []
  elseif <null[tree]> then []
  else <concat
        <[quickbatch hit quickbatch] [
          <ℓt-eq-gt[ℓ <key[tree]> ]>
                tree                    ]> > ;
<hit[ℓ info]> ≡
  if <null[ℓ]> then []
  else [info] .
```

The last line of <u>quickbatch</u> deserves some explanation. The result of the use of functional combination is three lists of associations on keys which are to be concatenated. The first and third are derived from recursive calls on the left and right subtrees of the non-null tree. The middle list is empty unless the key found at the root of the tree happened to be mentioned once or more in the target list of the search. Finally, the sorting of the answer list is carried out by an implicit Quicksort at each node in the search tree. The function lt-eq-gt partitions at <key[tree]> the target list carried in an unordered batch to tree. For example, if tree is



then <quickbatch[[9 2 3 6 8 7 3] tree]> evaluates to
((2 ant)(3 boa)(8 eel)(9 dor)) .

Stars

The next language feature is called "star" because of its syntax, reminiscent of the Kleene star. The list [A*] evaluates to the list (A*) = (A A A A ...), which has the semantics of a list of an infinite number of A's, although it may be represented in finite space and printed in finite (star) notation. Similarly, [0*] evaluates to an infinite list of zeros (the zero vector) which, fortunately, may be printed as (0*); [x*], under the binding of x as (2 4 6 8), evaluates to a matrix with an infinite number of rows and only four columns which may be printed: ((2 4 6 8)*) .

The star notation may be applied in constructing combinators if all elements are identical. For instance, in order to add one to every element of a vector, x, one can write

```
<[sum*][
  [ 1* ]
    x   ]>
```

which evaluates to (3 5 7 9) under our binding for x. The definition of functional combination above still applies under the convention that the values m_i can be infinity for starred rows. In the previous example $m_0 = \infty = m_1$ and $m_2 = 4$ so $m = 4$. Of course if all $m_i = \infty$, then $m = \infty$, as established by the convention

$$
\begin{aligned}
&<[f^*]\ [\\
&\quad [\alpha_1{}^*]\\
&\quad [\alpha_2{}^*]\\
&\quad \vdots\\
&\quad [\alpha_n{}^*]]> \;=\; [<f[\alpha_1\ \alpha_2\ \ldots\ \alpha_n]>^*]\ .
\end{aligned}
$$

The star notation may be applied only to the suffix of a list whose prefix is explicitly expressed: [cons cons sum*] is a legal combinator and [2 3 4 5*] evaluates to (2 3 4 5 5 5 ...).

Starred structures are most useful in the context of functional combination. Starred functions are "spread" (or mapped [20]) across all available columns of the argument matrix; starred arguments are shared by all columns. As an example of the impact of stars we present Gaussian matrix multiplication, leaving the definition of transpose to the reader.

```
<dotproduct[v1 v2]> ≡ <sum<[product*][
                                   v1
                                   v2    ]>>;
<row[vec transp]> ≡ <[dotproduct*][
                       [   vec*    ]
                           transp     ]> ;
<mtxmpy[m1 m2]> ≡ <[row*][
                       m1
                     [<transpose[m2]>*] ]> .
```

The role of suspending cons

The function cons is representative of an entire class of functions which build structures by filling in the values of fields within nodes. Syntactically it also serves as a space allocator although that characteristic plays a lesser role in the following discussion. We have proposed a new semantics for cons and its extractor functions first and rest which avoids the construction of those portions of structures that are never accessed after their creation. The results apply to any operation which assigns a value to a field, provided that it is possible to preserve a record of all relevant bindings. This criterion is difficult to meet in a system in which users can change assigned values, but it is easily satisfied under a regime of applicative programming in which the user can only create and implicitly release such bindings [15].

Using the function cons as a paradigm of structure-creating functions, we briefly explain its semantics. When cons is invoked by the user, the value returned is a pointer to a newly built structure. Rather than evaluate the arguments to cons and create the complete structure, we create a structure consisting of two suspensions. A suspension consists of a reference to the form whose evaluation was deferred and a reference to the environment of variable bindings in which the suspension was originally created. These two structures must remain intact for the life of the suspension. The reference to the form is a pointer to a piece of program, so the space it occupies usually represents no great overhead. Environments present more of a problem, since we are accustomed to viewing them only as temporary structures. Moreover, use of destructive assignment operations generally requires recreation of the entire environment in order to assure the integrity of references to the environment as it existed before the assignment. Destructive assignments, if not well controlled, become costly; it is fortunate that they do not exist in our source language.

When either of the functions first or rest is invoked, the following events occur. A designated field of the argument is checked to determine if it contains a suspension (suspensions are flagged and easily distinguished); if not, then its contents is returned. If a suspension is present, then the evaluator is invoked upon the designated form within the preserved environment. The result is stored back in the designated field in place of the suspension (for next time), and the value is returned as a final result. These events constitute coercion of the suspension. The two functions, first and rest, therefore act as probes into the data structure, with possible effects of a predictable and benign sort, rather than as simple extractor functions.

As a result of suspending, evaluations are delayed as long as possible. Ultimately all evaluations take place as a result of the demands of the driver of the output device which tries to move the contents of its list to the external device. As it traverses the list it is outputting, it invokes first and rest, causing top-level evaluation, which in turn results in the creation and inspection of more structure, indirectly forcing all of the necessary evaluations. Regardless of the intentions of the programmer, the only structures which are actually built are those which are essential to deciding what information is to be output. Least-fixed-point semantics for the language result [8].

A fortunate side-effect of suspending the creation of data structures is the ability to deal with infinite structures. Consider the list defined (but never completely constructed) by the invocation of <terms[0]> where

```
<terms[n]> ≡ <cons[<recip[<square[n]>]>
                   <terms[<addl[n]>]> ]> .
```

That list, the reciprocals of the squares of all the positive integers, might be familiar since its sum, excluding the first term, converges to $\pi^2/6$. Suppose that z were bound to the result of <terms[0]>; in fact, because of the suspending cons, z is initially bound only to a "promise" of this result. As long as <1 z> is not computed (since it diverges on division by zero) and as long as a complete traversal of the structure is not invoked, the infiniteness of z poses no problem. An access to <6 z>, if essential to the output device, would find the answer 0.04 even though that number had not been present before that access; it would have been computed had it been of interest earlier. (This use of cons is similar to Landin's prefixs [19] as explained by [3], but it differs precisely in that the rest of the list z may be accessed without computing the divergent first element.) More implications of cons on infinite structures may be found in [11].

The same techniques used for cons may be applied to any record creating (field assignment) function within the system. We have proposed an interpreter [8] in which all field assignments are suspended. This has a great impact, as in particular the construction of environments may be suspended. This means that no argument will be evaluated unless the corresponding formal parameter has been accessed by some operation critical to the execution of the program (i.e. critical to the creation of output). This effects the call-by-need argument-passing protocol [24], the call-by-delayed-value [23], and lazy evaluation scheme [12].

Another effect is on the semantics of functional combination. The result of an application of functional combination is a list which, not surprisingly, is conventionally built with cons. If cons suspends then only those items in that list which are accessed are ever created. For instance, the result of an invocation of quickbatch is a list. That list, if not trivial, is the result of an invocation of concat which uses cons. Later arguments to concat need not all be computed at once (or even at all if only a part of the result were ever needed for printing). The argument list for concat is the result of functional combination and thus, as we suggest here and demonstrate elsewhere [6, 7] need not be computed all at once. Instead of computing the complete answer, only that computation path essential to the answer is pursued. Intermediate environments are preserved in case any suspensions are coerced later. Recursive calls on the left and the right subtrees often need not both be evaluated. For instance, only five recursive calls on quickbatch are required to determine the first information-pair, (2 ant), in the example above which requires fourteen recursive calls (plus the outermost call) in order to ascertain the final answer.

Opportunities for parallelism

With the language defined, we now turn to the opportunities for parallelism provided in the language. We do not explicitly require these parallelisms to be performed, nor do we require that the programmer be aware that they even may occur. Programs are easily written with these control structures with the semantics described in the previous section, which do not depend on concurrencies. It is significant that some of the semantics of the language allow for improvements in parallel interpretation of programs written in a very popular language differing only syntactically from a part of ours [20]. It is the role of a compiler to detect the opportunities for parallelism in its pass over the program before runtime and to alter the code to be interpreted in order to provide for the parallelism allowed by the target hardware. The responsibilities for synchronization are therefore the concern of the compiler so the programmer need not worry about issues of "structured multiprogramming" [2].

At the same time that we say that the compiler should detect parallelism for runtime, we should point out how the source language helps the compiler in this task by allowing simple program structures. Most notably, the language does not have destructive assignment statements; it is free of side-effects. All variable/value bindings are established as parameter/argument bindings in function linkages, and they are therefore not subject to change during their lifetime. An obvious (but not new [25]) opportunity for parallelism is collateral argument evaluation, establishing these bindings simultaneously since they are independent of one another. These bindings are abandoned after all computations under the environment of the function invocation have been completed, but until then they remain intact. This integrity of environments, essential to the suspending cons, also alleviates the concurrency problem, since no conflict arises because of a reader accessing a value as a writer alters it [4].

The feature of suspending cons, itself, provides opportunity for massive parallelism. A system implemented with only the user's invocations of cons suspended, or with those and all the system structures suspended, may have hundreds of suspensions pending on the system during the course of computation. In a single processor system all (but one) of these would await probing by the system functions, first and rest, before their coercion would be initiated. If the runtime environment were enriched with idle processors, then any of these suspensions could be coerced simultaneously without delaying the progress of the critical evaluation (the single one active on a single processor). Let us designate that distinguished evaluation as the colonel and any other processors available will be called sergeants.

The parallel evaluation strategy is to keep the colonel working on the same critical process which would occupy a single processor and to allocate the sergeants to suspensions which are "near" the colonel process. Since evaluation of suspensions usually converges to nodes containing new suspensions rather quickly, sergeants tend to finish tasks rapidly after which they are reassigned to new ones "closer" to the moving colonel. (It is possible that a sergeant could fall into a divergent evaluation and therefore be lost to the system until the suspension it was evaluating becomes irrelevant.) The colonel behaves exactly as a single processor would, except that from time to time it accesses what would have been a suspension and instead finds the result already provided by a sergeant who had passed through earlier. The definition of the "near" metric should be chosen to maximize the likelihood of this fortunate event. The sergeants scurry about the system following the colonel doing their best to satisfy his anticipated needs. Some of their effort may be wasted since not all handiwork of sergeants need be accessed by the colonel. Yet the time to compute the final result is no more than the time using a single evaluator since parallelism has been provided at essentially no overhead. There is no cost due to interprocessor conflict and communication. Some additional cost may arise from the enforcement of the "near" metric; but this requires overhead only as a sergeant process is initiated -- not while it's running.

Even though a processor has been led down the gardenpath (diverges) [8], there is still an opportunity for recovery, if the value it is supposedly computing is discovered to have become unnecessary to the system. This, in fact, is rather easily accomplished because processor allocation is so closely tied to the data structure. The same mechanism which determines that a node has become useless and is to be returned to available space need only stop execution of any process (some wayward sergeant) which is operating on a suspension referenced from that node. Since all space allocated by the colonel for its computation will be returned after the result has been provided, it follows that all sergeants will be recovered as well by that time. Therefore, if the colonel's computation converges it is not possible to lose a sergeant; all space and processors will be restored to the system.

Functional combination offers two sorts of parallelism. The first is exemplified by the code for ℓt-eq-gt. In the definition for this function the recursion is linear down the list parameter, but at each recursion step each of the three developing results must be handled. Clearly the three pieces of the ultimately final result can be handled by three concurrent processes. So a simple but bounded parallelism is provided depending on the size (m in the definition above) of the result when all elements of the combinator are defined independently of the function definition in which the combinator appears.

Another kind of parallelism results if that function itself appears in the combinator. The coding of the function quickbatch is an example of this. If m processors are allocated for computing the result of a combinator and a combinator has occurrences of the function being defined as some f_j, then a process tree can result with processors active only at the leaves. The tree results because a single processor evaluating a recursive function might encounter an instance of functional combination and become dormant while the m processes from that instance compute. If some of those processes are recursive invocations, then each of those processes may become dormant in the same way. If all processes terminate then the invocation tree is of finite depth with degree m at any node, with dormant processes at all non-leaves, and with active processes only at the leaves. If a combinator has more than one recursive call in such a scheme then a very "bushy" process tree can result. For example, the quickbatch function of the Quicksort Algorithm can be implemented so that every recursion requires a new processor. At the n^{th} level 2^n processors may be required. The processors are all evaluating the same function definition under disjoint (and static) environments, however, so that lock-step evaluation is entirely appropriate.

These semantics require very little interprocessor protocol. Upon interpretation of functional combination the active process goes dormant and spawns m new processes. Each of these processes is independent and need not initiate communication with any other user process except to report its result. As it reports its result a process dies but its dormant parent is jarred; we call this process stinging. A stung parent becomes active when it is stung with the (chronologically) last result. Therefore, the only run-time processor synchronization involves process creation and stinging. (Environments are static!) This is no more complicated than what is required for collateral argument evaluation.

The star notation used on an argument to functional combination merely denotes that the argument is to be shared by all m processors. When the combinator itself is a starred structure then the combinator is implicitly homogeneous and a different sort of concurrency may be used for interpreting the function. This use of combinators is most similar to mapping functions [20] and their generalization [22]. An example is the code for dotproduct above in which all additions may take place concurrently. Due to the expression of the combinator with the star, the compiler can easily detect that the same operation will be performed on all objects in the data structures which are arguments to the combinator. Then the evaluation may proceed using pipelining across the n arguments to the starred combinator.

Similarly, the starred notation used within the combinator itself denotes that the code for the function is to be used by each of the m processors. Under parallel interpretation this kind of functional combination has the semantics of shared pure code. For instance, the algorithm for mtxmpy specifies that the code for the function row is to be shared by all processors used in interpreting its functional combination, up to M in an M by N times an N by P problem. Also, row specifies that the code for dotproduct can be shared by the up to N processors used for its functional combination, where each of these is a starred combinator distributing the code for the primitive instruction product across P processors. Thus up to N×M×P multiplications might be performed simultaneously by processors interpreting the shared code in parallel.

Conclusions

Functional combination allows the use of known forms of controlled parallelism, whereas the suspending cons will allow masses of sergeant processors to be occupied on heuristically useful computation. The former facility fits existing hardware which now requires specific higher-level languages and specially trained programmers in order to occupy the processors productively. The latter approach offers a hope for occupying a machine with arbitrarily large numbers of processors whose temporal configuration cannot be known to the programmer.

This ability of our semantics to use a system with massive parallelism (thousands of processors) is very important for future hardware design. Such systems will not be built unless there is a way to program them, even though the current cost of processors suggests that they will be technically possible. With communication cost high and processor cost negligible, pressure will build for a massive computation on data while it remains within storage directly accessible to any processor. Not only do our semantics admit such massive (albeit heuristic) parallelism, but also they achieve these results on a well known language, pure LISP, imposing these semantics on programs extant fifteen years ago.

Taken together these approaches to programming in purely applicative source code provide the programmer with higher-level tools for expressing algorithms so that the compiler can recognize and compile parallel code.

Acknowledgement: We are grateful for the suggestions of C. Brown which helped refine this paper. S. Smoliar pointed out the implications of functional combination for sharing code.

References

[1] J. Backus, Programming language semantics and closed applicative languages, Proc. ACM Symposium on Principles of Programming Languages (1973), 71-86.

[2] P. Brinch Hansen, Concurrent programming concepts, Computing Surveys 5, 4 (1973), 223-245.

[3] W. H. Burge, Recursive Programming Techniques, Addison-Wesley, Reading, MA (1975).

[4] P.J. Courtois, F. Heymans, and D. L. Parnas, Concurrent control with 'readers' and 'writers', Comm. ACM 14, 10 (1971), 667-668.

[5] D. P. Friedman, and D. S. Wise, Unwinding stylized recursions into iterations, Computer Science Department, Indiana University, Technical Report No. 19 (1975).

[6] D. P. Friedman, and D. S. Wise, Multiple-valued recursive procedures, Computer Science Department, Indiana University, Technical Report No. 27 (1976).

[7] D. P. Friedman, and D. S. Wise, An environment for multiple-valued recursive procedures, Proc. 2nd Colloque sur la Programmation, Springer-Verlag, Berlin (1976).

[8] D. P. Friedman, and D. S. Wise, CONS should not evaluate its arguments. In S. Michaelson & R. Milner (eds.), Automata, Languages and Programming, Edinburgh University Press, Edinburgh (1976), 257-284.

[9] D. P. Friedman, and D. S. Wise, Output driven interpretation of recursive programs, or writing creates and destroys data structures, Computer Science Department, Indiana University, Technical Report No. 50 (1976).

[10] D. P. Friedman, D. S. Wise and C. A. Brown, Implementation of extended semantics for pure LISP (in preparation).

[11] D. P. Friedman, D. S. Wise and M. Wand, Recursive programming through table look-up, Proc. ACM Symp. on Symbolic and Algebraic Computation (1976), 85-89.

[12] P. Henderson, and J. Morris, Jr., A lazy evaluator, Proc. 3rd ACM Symp. on Principles of Programming Languages (1976), 95-103.

[13] C. E. Hewitt, and B. Smith, Towards a programming apprentice, IEEE Trans. on Software Engineering SE-1, 1 (1975), 26-45.

[14] C. A. R. Hoare, Quicksort, Computer J. 5, 1 (1962), 10-15.

[15] C. A. R. Hoare, Towards a theory of parallel programming. In C.A.R. Hoare & R.H. Perrott (eds.), Operating Systems Techniques, Academic Press, London (1972), 61-71.

[16] G. Kahn, The semantics of a simple language for parallel processing, Proc. IFIP Congress, North-Holland, Amsterdam (1974), 471-475.

[17] D. E. Knuth, Sorting and Searching, Addison-Wesley, Reading, MA (1973).

[18] D. E. Knuth, Fundamental Algorithms (2nd ed.), Addison-Wesley, Reading, MA (1975).

[19] P. J. Landin, A correspondence between ALGOL 60 and Church's lambda notation, Part I., Comm. ACM 8, 2 (1965), 89-101.

[20] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, MA (1962).

[21] B. Shneiderman, and V. Goodman, Searching of sequential and tree structured files, ACM Transactions on Database Systems 1, 8 (1976).

[22] G. Tesler, and H. J. Enea, A language design for concurrent processes, Proc. Spring Joint Computer Conference, Thompson, Washington (1968), 403-408.

[23] J. Vuillemin, Correct and optimal implementation of recursion in a simple programming language, J. Comp. Sys. Sci. 9, 3 (1974), 332-354.

[24] C. Wadsworth, Semantics and Pragmatics of Lambda-calculus, Ph.D. dissertation, Oxford (1971).

[25] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, Revised report on the algorithmic language ALGOL 68, Acta Informatica 5, 1-3 (1975), 1-236.

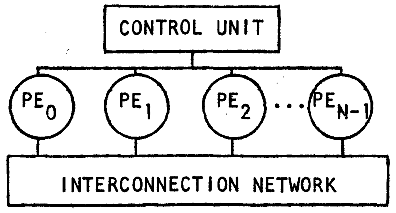
SINGLE INSTRUCTION STREAM - MULTIPLE DATA STREAM MACHINE INTERCONNECTION NETWORK DESIGN[*]

Howard Jay Siegel[**]
Department of Electrical Engineering
Princeton University
Princeton, New Jersey 08540

Abstract -- An SIMD machine must have an interconnection network to pass data between processing elements. We introduce a model of SIMD machines which allows a formal mathematical analysis and comparison of different interconnection networks. Five interconnection networks that have been proposed in the literature are defined in terms of our model. They include a network similar to the one used in the STARAN, a network similar to the one recommended by Feng to implement data manipulating functions, the Illiac IV network, and the Perfect Shuffle. The networks are evaluated in terms of the upper and lower bounds on the time required for each network to simulate the actions of the others. It is usually impractical to implement all the interconnections that may be needed by the machine to perform a large variety of computations, so the ability of a network to simulate other interconnections is important. The methods used to prove the lower bounds and to construct the simulation algorithms to demonstrate the upper bounds can be generalized and applied to the analysis of other networks.

I. Introduction

One aspect of the design of SIMD (single instruction stream - multiple data stream [8]) or array machines is the construction of an interconnection network to pass data from one processor to another. One way to view the structure of an SIMD machine is as a set of N processing elements (where each processing element consists of a processor with its own memory), interconnected by a network, and fed instructions by a control unit.



The network connects each processing element to some subset of the other processing elements. The connections are represented by a set of interconnection functions. Only one interconnection function of the network can be used at a time; i.e., at any time, each processing element is connected to only one other processing element. A transfer instruction causes data to be moved from each processing element to the processing element to which it is connected. To move data between two processing elements that are not directly connected, the data may be passed through intermediary processing elements by executing a programmed sequence utilizing the interconnection functions in that network.

When building an SIMD machine, an interconnection network must be implemented. To choose which interconnection functions to include in the network, the system designer must consider the types of problems the machine will be used to solve. Generally, it is not possible to include all of the interconnection functions desired. Therefore, those that will be used most often would be implemented and used to simulate the other interconnections that may be required. Also, an SIMD machine may be being designed as a general purpose machine, to handle a large variety of tasks. Thus, it is very important for the system designer to consider the ability of a set of interconnection functions to simulate other interconnection functions.

In this paper we shall develop a realistic model of SIMD machines and use it to evaluate interconnection networks. We shall discuss five particular interconnection networks and show them to be equivalent in the sense that each can simulate the actions of the others. The networks we will examine are: the Cube network, a network similar to the one implemented in the STARAN machine [3]; the PM2I network, a network similar to the one used by Feng to implement data manipulating functions [6], [7]; the Illiac network [1], [5]; the Perfect Shuffle, which has been popularized by Stone [18]; and the WPM2I network, a variation of PM2I which was introduced in [14]. These networks are analyzed in terms of the time complexity required for one network to simulate another.

A model independent lower time bound for each simulation shall be presented. Many of these lower bounds are proved in [14]. In this paper we shall prove only those lower bound results which are better than those presented in [14].

[*]This work was supported by NSF Grant DCR74-21939. The article is a revised summary of Princeton University, Department of Electrical Engineering, Computer Science Laboratory Technical Report 198.

[**]Author's current address is Purdue University, School of Electrical Engineering, West Lafayette, Indiana 47907.

The upper time bound for each simulation shall be demonstrated by an algorithm that performs the simulation. The methods used to construct these algorithms can also be used to write algorithms to simulate interconnections not presented here. The algorithms we shall present can be directly implemented on an SIMD machine that satisfies the assumptions we shall make in section IV.

II. The Model

Our model of an SIMD machine consists of four parts: processing elements, interconnection network, machine instructions, and masking schemes. Each processing element (PE) is a processor together with its own memory, a set of at least three fast access registers (A, B, and C), and a data transfer register (DTR). The DTR of each PE is connected to the DTR's of the other PE's via the interconnection network. When an interconnection function is executed, it is the DTR contents of each PE that are transferred.

There are N PE's, each assigned an address from 0 to N-1. We assume that $N = 2^m$; i.e., $\log_2 N = m$. We also assume that PE_i has a register ADDRESS that contains the integer i. Let ADDRESS(j) be the jth bit of ADDRESS.

Each PE is always in either active or inactive mode. If a PE is active it executes the instructions broadcast to it by the control unit. If a PE is inactive it will not execute the instructions broadcast to it.

An interconnection network is a set of interconnection functions, each a bijection on the set of PE addresses. When an interconnection function f is applied, PE_i copies the contents of its DTR into the DTR of $PE_{f(i)}$. This occurs for all i simultaneously, for $0 \leq i < N$ and PE_i active. Thus, saying an interconnection network maps the address x to the address y is equivalent to saying that it causes PE_x to pass its data to PE_y. Note that an inactive PE may receive data from another PE if an interconnection function is executed, but it cannot send data.

To pass data from one PE to another PE a programmed sequence of interconnection functions must be executed. This sequence of functions moves the data from one PE's DTR to another's by a single transfer or by passing the data through intermediary registers.

For example, let one of the interconnection functions f in a network be defined by $f(x) = (x+1) \bmod N$, where x is a PE address. Then when f (the cycle function) is applied, PE number 0 transfers the contents of its DTR to the DTR of PE number $f(0) = 1$, PE number 1 transfers the contents of its DTR to the DTR of PE number $f(1) = 2, \ldots$, and PE number N-1 transfers the contents of its DTR to the DTR of PE number $f(N-1) = 0$. To pass data from PE_i to $PE_{i+2 \bmod N}$, $0 \leq i < N$, f may be executed tiwce.

In section III five particular interconnection networks will be defined.

The machine instructions are those operations that each processor can perform on data in its individual memory or registers. We assume there is a separate control unit (CU) computer which stores programs and broadcasts instructions and data. All active PE's execute the same instruction at the same time, but on possibly different data.

Actual SIMD machines allow data to be moved among the memory, the fast access registers, and the DTR of a single PE. All we assume, without loss of generality, is that data may be moved among the registers. The notation $X \leftarrow Y$ means the contents of register Y are copied into register X, where X and Y could be A, B, C, or DTR.

A masking scheme is a method for determining which PE's will be active at a given point in time. An SIMD machine may have several different masking schemes. Each mask partitions the set of PE addresses into those PE's that will be active and those that will be inactive.

If PE address masks are used, an m-position mask will accompany each instruction and will determine which PE's are active, i.e., execute that instruction. Each position of the mask is either a 0, 1 or X ("don't care"), and the only PE's that will be active are those that match the mask for each of the m bit positions of their address. For example, if N = 8 (so m = 3) and the mask is 1X0, then only PE's 6 (110) and 4 (100) would be active. Superscripts will be used as repetition factors, e.g., $X^3 01^2$ would be XXX011. Square brackets will be used to denote a mask. For example, executing the instruction "$DTR \leftarrow A\ [X^{m-1}0]$" would cause each even numbered PE to load its DTR with the contents of its A register. This scheme was presented and discussed in [14].

Data conditional masks are the implicit result of executing "if-then-else" statements that involve local data in each PE's registers or memory. This type of masking is used in such machines as the Illiac IV ([1], [5], [12]) and PEPE [20]. Whenever a conditional statement is executed each PE may be executing it with different data, so the outcome may differ from one PE to the next. Thus, as a result of the conditional each PE will set an internal flag so that it will be active for either the "then" or the "else," but not both. The execution of the "else" statements must follow the "then" statements; i.e., they cannot be executed simultaneously. For example, as a result of executing the statement: "if $A > B$ then $C \leftarrow A$ else $C \leftarrow B$" each PE will load its C register with the maximum of its A and B registers; i.e., some PE's will execute "$C \leftarrow A$," and then the rest will execute "$C \leftarrow B$." Thus, for SIMD machines, data conditional masks and "if-then-else" statements are the same.

PE address masks and data conditional masks will be the only masking schemes used in this paper. PE address masks provide a concise method to activate 3^m different sets of PE's. Data conditional statements are an essential part of all programming languages, so it is fair to assume they would be present in all SIMD machines. The results of this paper would still be valid even if only data conditional masks were used. This is because if each PE knows its own address, then data conditional masks could be used to simulate PE address masks using no additional interprocessor data transfers.

Whenever an interconnection function is executed, all active PE's pass their data at the same time. Since each interconnection function is a bijection, this transfer of data occurs without conflict if all PE's are active. It is possible, however, that masking can cause transfers of data no longer to represent bijections on the PE addresses. Such data transfers would destroy data.

For example, let N = 8 and let the interconnection function be f(x) = (x+1) mod 8. Suppose f is executed with the PE address mask [0XX]. Then f(4) = 4, since PE number 4 is not active, and f(3) = 4, since PE number 3 is active. Thus, this data transfer is not a bijection, so it destroys data. In this case the original contents of the DTR of PE number 4 is destroyed by the data transferred into that DTR from PE number 3. In order to have saved this data the DTR contents of PE number 4 would have to have been copied into a register or memory location of that PE before the data transfer instruction was executed.

Formally, an SIMD machine can be represented as the 4-tuple (N,F,I,M), where:

(1) N is a positive integer, representing the number of processing elements in the machine;

(2) F is the interconnection network, where each interconnection function is a bijection on the set {0,1,...N-1};

(3) I is the set of machine instructions, instructions that are executed by each active PE and act on data within that PE;

(4) M is the set of masking schemes, where each mask partitions the set {0,1,...N-1} into the set of active PE's and the set of inactive PE's.

By specifying N, F, I, and M, a particular SIMD machine architecture can be modeled.

III. Interconnection Networks

Let the binary representation of a PE address be $p_{m-1}p_{m-2}\cdots p_1p_0$, let $\overline{p}_i$ be the complement of of p_i, and let the integer n be the square root of N.

(1) Perfect Shuffle (PS). This network consists of a shuffle function and an exchange function. The shuffle is defined by:

$$s(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-2}p_{m-3}\cdots p_1p_0p_{m-1}$$

and the exchange is defined by:

$$e(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-1}p_{m-2}\cdots p_1\overline{p}_0.$$

The shuffle function is a left rotation of the bits of each address. The exchange function complements the low order (0th) bit of each address. For example, s(3) = 6 and e(6) = 7, for N = 8. The shuffle can be thought of as the result of perfectly shuffling a deck of cards (i.e., 0 → 0, N/2 → 1, 1 → 2, N/2+1 → 3, etc.) (see [9], [10], [14], [18]).

This network has been shown to be quite useful by Stone in [18]. It is also the basis of Lawrie's "omega" network [11].

(2) Illiac. This network has four functions defined as follows (recall n is the square root of N):

$$I_{+1}(x) = x+1 \bmod N$$

$$I_{-1}(x) = x-1 \bmod N$$

$$I_{+n}(x) = x+n \bmod N$$

$$I_{-n}(x) = x-n \bmod N.$$

For example, if N = 16, $I_{+n}(0) = 4$. When we discuss the Illiac we shall assume m is even, that is, $n = 2^{m/2}$ is an integer. If the PE's are considered as a n x n array, then each PE will be connected to its north, south, east and west neighbors (see [1], [5], [12], [14], [17]).

This network is implemented in the Illiac IV system. The ability of this system to perform various tasks is described in [5].

(3) Cube. This network consists of m functions defined by:

$$c_i(p_{m-1}\cdots p_{i+1}p_ip_{i-1}\cdots p_0)$$

$$= p_{m-1}\cdots p_{i+1}\overline{p}_ip_{i-1}\cdots p_0$$

for $0 \leq i < m$. The Cube function c_i complements the ith bit of each address. For example, $c_2(7) = 3$. When the PE addresses are considered as the corners of an m-dimensional cube this network connects each PE to its m neighbors (see [14]). Note that c_0 and the Perfect Shuffle exchange function e are identical.

The network used in the STARAN is a wired series of Cube functions (see [3]). In [2] and [4] the applicability of this network to practical problems is discussed. A version of this type of network was used as part of a parallel machine simulation in [13].

(4) Plus-Minus 2^i (PM2I). This network consists of 2m functions defined by:

$$t_{+i}(j) = j+2^i \bmod N$$

$$t_{-i}(j) = j-2^i \bmod N$$

for $0 \leq i < m$. For example, $t_{+1}(2) = 4$ if $N > 4$. Note that the Illiac IV is a subset of this network. Various properties of the PM2I network can be found in [14].

The network recommended by Feng to implement data manipulating functions is a wired series of PM2I functions [6]. The various data manipulating functions that this network can perform are discussed in [6] and [7].

(5) Wrap-around Plus Minus 2^i (WPM2I). This network consists of 2m functions defined by:

$$w_{+i}(p_{m-1}\cdots p_i \cdots p_0) = q_{m-1}\cdots q_i \cdots q_0,$$

where

$$q_{i-1}\cdots q_0 q_{m-1}\cdots q_{i+1} q_i =$$
$$(p_{i-1}\cdots p_0 p_{m-1}\cdots p_{i+1} p_i)+1 \bmod N,$$

and

$$w_{-i}(p_{m-1}\cdots p_i \cdots p_0) = q_{m-1}\cdots q_i \cdots q_0,$$

where

$$q_{i-1}\cdots q_0 q_{m-1}\cdots q_{i+1} q_i =$$
$$(p_{i-1}\cdots p_0 p_{m-1}\cdots p_{i+1} p_i)-1 \bmod N,$$

for $0 \leq i < m$. WPM2I is like PM2I, except any "carry" or "borrow" will "wrap-around" to the p_{i-1} bit position. Note that any "carry" or "borrow" cannot affect p_i. For example, if $N = 8$ and $m = 3$, then $w_{-1}(001) = 110$, whereas $t_{-1}(001) = 111$.

The WPM2I network was introduced in [14]. It is a variation of the PM2I which has the ability to simulate any other interconnection function when the networks are treated as sets of permutations on the integers from 0 to N-1. In terms of group theory, WPM2I can generate the entire group of permutations on N elements. Of the five networks presented here, only WPM2I has this ability (see [14]).

IV. Simulations Results

The designers of SIMD machines must choose a set of interconnection functions to implement, and they will either base their choice on the type of computations the system will be expected to perform or assume they are building a general purpose machine. The number of functions that will be included in the network will be constrained by such factors as cost and hardware complexity. Therefore, it is quite important for the designer to consider the ability of the network that is chosen to simulate other functions.

In this section we compare the simulation ability of five different types of interconnection networks that have been proposed in the literature and have been shown to be useful. The lower bounds on simulation times and the simulation algorithms that follow demonstrate techniques that can be used to compare and analyze other networks.

We use these specific simulations to demonstrate our methods for several reasons. There is little in the literature directly comparing the abilities of these types of networks. The following theorem provides a means for such a comparison. In addition, by using these simulations to demonstrate the techniques, the system designer may observe the minimum number of data transfers needed if a network presented here was implemented and it was then found necessary to simulate the actions of one of the networks that we have defined. The designer is also provided with an algorithm to perform the simulation. Since these networks have been shown to be useful it is very possible that any network implemented may have to simulate one of them.

The lower bound results are valid for all models of SIMD machines. The only assumptions made for Theorem 1 are:

(1) that at any given point in time a PE must be either active or inactive;

(2) that the interconnection function, of the network to be simulated, which requires the most time to simulate, will determine the lower time bound for the network; and

(3) that when an interconnection function is simulated, its effect on all PE's must be simulated.

The model - independence is significant because it means that the results and the methods used to obtain them apply to real machines.

The lower bounds are in terms of the number of times interconnection functions must be executed in order to perform the simulation. Recall that the transferring of data from PE_x to PE_y will also be referred to as mapping the address x to y. The mappings will be described by logical or arithmetic operations on the m bits of the PE addresses.

Theorem 1 explores the lower bounds on the time required for each network defined in section III to simulate the other networks. In Theorem 2 the upper bounds on the simulation time are analyzed.

Many of the lower bound results were presented in [14]. We will sketch the proofs of only those new results which provide tighter bounds.

Theorem 1: In the following table the entry in row x, column y, is a lower time bound for network x to simulate network y. An * indicates that the proof of the bound is sketched in [14].

	Cube	PS	Illiac	PM2I	WPM2I
Cube	-	$2\lfloor m/2 \rfloor$*	m*	m*	m*
PS	m+1	-	2m-1*	2m-1*	2m-1*
Illiac	(n/2)+1	(n/2)+1*	-	n/2*	(n/2)+1*
PM2I	2	m	1	-	2
WPM2I	2	m/2*	3	3	-

Proof: The notation "x → y" means "the case where x is used to simulate y."

PS → Cube: Observe that $c_1(1^{m-2}01) = 1^m$ and $c_1(1^m) = 1^{m-2}01$. At least m-1 shuffles must be executed to map $1^{m-2}01$ to 1^m, as we must move the 0 to the 0th bit position so that the exchange can change it. The only way to perform this mapping in m steps is to execute m-1 shuffles followed by one exchange. To map 1^m to $1^{m-2}01$ at least one shuffle must be used after an exchange is executed. Therefore, at least m+1 steps are required.

Illiac → Cube: Let $d(x,y) = |x-y|$, the absolute difference of x and y. The function d is a metric (see [14]). Let $j = (m/2)-1$. $d(0,c_j(0)) = n/2$. $d(x,I_{+n}(x)) = d(x,I_{-n}(x)) = n$, $0 \leq x < N$, so I_{+n} and I_{-n} cannot be used to move a distance of n/2. $d(x,I_{+1}(x)) = d(x,I_{-1}(x)) = 1$, $0 \leq x < N$. Therefore, the only way to map 0 to n/2 in n/2 steps is to execute I_{+1} n/2 times. But $c_j(1^m) = 1^{m/2}01^{(m/2)-1}$ and no subsequence of $(I_{+1})^{n/2}$ can perform this mapping. Thus, at least (n/2)+1 steps are required.

PM2I → Cube: For $0 \leq j < m-1$ and $0 \leq i < m$ $c_j \neq t_{\pm 1}$. Thus, at least two steps are required.

PM2I → PS: An interconnection function f has the effect of adding x distinct integers, mod N, to x different addresses, one to each address if $f(k_i)-k_i = q_i$, such that $k_i \neq k_j$ and $q_i \neq q_j$ for $i \neq j$, $0 \leq i, j < x$. Each execution of a PM2I function can add either a mod N integer, if the PE is active, or nothing, if the PE is inactive, to the set of PE addresses. Thus, the number of distinct integers added to addresses after $\log_2 x$ executions of distinct PM2I functions is at most x. The shuffle function has the effect of adding N-1 distinct integers to the set of addresses. Thus, the PM2I network requires at least $\lceil \log_2(N-1) \rceil = m$ steps to simulate the shuffle.

PM2I → Illiac: $I_{\pm 1} = t_{\pm 0}$, $I_{\pm n} = t_{\pm(m/2)}$.

PM2I → WPM2I: For $0 < j < m$ and $0 \leq i < m$, $w_{\pm j} \neq t_{\pm i}$. Thus, at least two steps are required.

WPM2I → Cube: For $0 \leq j < m$ and $0 \leq i < m$, $c_j \neq w_{\pm i}$. Thus, at least two steps are required.

WPM2I → Illiac: $I_{+n}(1^m) = 0^{m/2}1^{m/2}$. The only way WPM2I can perform this mapping in two steps is w_{-0} followed by $w_{+(m/2)}$. $I_{+n}(1^{m/2}0^{m/2}) = 0^m$. The only way WPM2I can perform this mapping in two steps is $w_{+(m/2)}$ followed by w_{-0}. Thus, at least three steps are required.

WPM2I → PM2I: Follows from WPM2I → Illiac analysis. □

In Theorem 2 we demonstrate methods to construct algorithms to simulate particular interconnections. (In [16] algorithms to simulate arbitrary interconnections are presented.) The algorithms that follow have more than theoretical significance. Given an SIMD machine which satisfies the assumptions we will make these algorithms can actually be used to perform the various simulations.

We make the following assumptions:

(1) All results are in terms of the model presented in section II, where $N = 2^m$, F will vary, I includes instructions for moving data between the DTR and the other registers of the same PE, and M = {PE address masks, data conditional masks}. (Recall that data conditional statements can be used to simulate PE address masks without using any additional interprocessor data transfers.)

(2) Time bounds are in terms of the number of executions of interconnection functions.

(3) When simulating the interconnection function f the data to be transferred

starts in the DTR of PE_x and must end in the DTR of $PE_{f(x)}$, $0 \leq x < N$.

(4) The interconnection function is to be simulated as if it were executed with all PE's being active. In [15] it is shown how this restriction can be removed.

When PE address masks and data conditional masks are used together, the PE address masks accompany each instruction in the "then" block and in the "else" block. Thus, in order for a PE to be active it must be in active mode as a result of the conditional and match the PE address mask accompanying the instruction. The notation $A \leftrightarrow B$ is an abbreviation for registors A and B switching their contents using a third register.

After each algorithm an example is given to demonstrate how the algorithm operates. For the examples we assume that the original contents of the DTR of PE_i is the integer i, all addresses and integers will be in binary, and unless otherwise stated, N will equal 8.

Theorem 2: In the following table the entries in row x, column y, are lower and upper bounds on the time required for network x to simulate network y given the above assumptions. Each upper bound is based on the time complexity of the algorithm presented to do that simulation.

	PM2I	PS	Cube	WPM2I	Illiac
PM2I					
lower	-	m	2	2	1
upper	-	2m-2	2	2	1
PS					
lower	2m-1	-	m+1	2m-1	2m-1
upper	2m	-	m+1	2m	2m
Cube					
lower	$2\lfloor m/2 \rfloor$	m	-	m	m
upper	m	m	-	m	m
WPM2I					
lower	3	m/2	2	-	3
upper	3	2m-2	2	-	3
Illiac					
lower	n/2	(n/2)+1	(n/2)+1	(n/2)+1	-
upper	n/2	6(n-1)	(n/2)+1	(n/2)+1	-

Proof: The notation "x → y" means "the case where x is used to simulate y." In [15] we discuss each algorithm and prove that it is correct.

PM2I → PS: For the exchange see the PM2I → Cube analysis, since $c_0 = e$.

For the shuffle:

```
(S1) A ← DTR [$1X^{m-1}$]
(S2) for i = m-2 until 0 step -1 do
         $t_{+i}$ [$X^{m-(i+2)}01X^i$]
(S3) B ← DTR [$X^{m-1}0$]
(S4) DTR ← A [$1X^{m-1}$]
(S5) for i = m-2 until 0 step -1 do
         $t_{-i}$ [$X^{m-(i+2)}10X^i$]
(S6) DTR ← B [$X^{m-1}0$]
```

PE	DTR	S1 A	S2 i=1 DTR	S2 i=0 DTR	S3 B	S4 DTR	S5 i=1 DTR	S5 i=0 DTR	S6 DTR
000	000	-	000	000	000	000	000	000	000
001	001	-	001	001	-	001	001	100	100
010	010	-	010	001	001	001	100	100	001
011	011	-	011	011	-	011	101	101	101
100	100	100	010	010	010	100	100	100	010
101	101	101	011	011	-	101	101	110	110
110	110	110	110	011	011	110	110	110	011
111	111	111	111	111	-	111	111	111	111

Example of PM2I → Shuffle

PM2I → Cube: For c_{m-1} use $t_{+(m-1)}$. For c_i, $0 \leq i < m-1$:

```
(S1) $t_{+i}$ [$X^m$]
(S2) $t_{-(i+1)}$ [$X^{m-(i+1)}0X^i$]
```

PE	DTR	S1 DTR	S2 DTR
000	000	110	010
001	001	111	011
010	010	000	000
011	011	001	001
100	100	010	110
101	101	011	111
110	110	100	100
111	111	101	101

Example of PM2I → c_1

PM2I → WPM2I: For w_{+0} use t_{+0} and w_{-0} use t_{-0}. For w_{+i}, $0 < i < m$ (w_{-i} similar):

```
(S1) A ← DTR [$0^m$]
(S2) $t_{+0}$ [$1^{m-i}X^i$]
(S3) A ↔ DTR [$0^m$]
(S4) $t_{+i}$ [$X^m$]
(S5) DTR ← A [$0^m$]
```

PE	DTR	S1 A	S2 DTR	S3 A	S3 DTR	S4 DTR	S5 DTR
000	000	000	111	111	000	110	111
001	001	-	001	-	001	110	110
010	010	-	010	-	010	000	000
011	011	-	011	-	011	001	001
100	100	-	100	-	100	010	010
101	101	-	101	-	101	011	011
110	110	-	110	-	110	100	100
111	111	-	110	-	110	101	101

Example of PM2I → w_{+1}

PM2I → Illiac: $I_{+1} = t_{+0}$, $I_{-1} = t_{-0}$, $I_{+n} = t_{+(m/2)}$, $I_{-n} = t_{-(m/2)}$.

PS → PM2I: For t_{+i}, $0 \leq i < m$ (t_{-i} is similar):

```
(S1) for j = i until m-1 do
(S2)      s [$X^m$]
(S3)      e[$1^{m-(j+1)}X^{j+1}$]
(S4) for j = 1 until i do   s[$x^m$]
```

PE	DTR	S2 j=1 DTR	S3 j=1 DTR	S2 j=2 DTR	S3 j=2 DTR	S4 j=1 DTR
000	000	000	000	000	110	110
001	001	100	100	110	000	111
010	010	001	001	100	010	000
011	011	101	101	010	100	001
100	100	010	110	001	111	010
101	101	110	010	111	001	011
110	110	011	111	101	011	100
111	111	111	011	011	101	101

Example of PS → t_{+1}.

PS → Cube: For c_0 use the exchange function e. For c_i, $0 < i < m$:

```
(S1) for j = 1 until m-i do   s[$X^m$]
(S2) e [$X^m$]
(S3) for j = 1 until i do   s[$X^m$]
```

PE	DTR	S1 j=1 DTR	S1 j-2 DTR	S2 DTR	S3 j=1 DTR
000	000	000	000	010	010
001	001	100	010	000	011
010	010	001	100	110	000
011	011	101	110	100	001
100	100	010	001	011	110
101	101	110	011	001	111
110	110	011	101	111	100
111	111	111	111	101	101

Example of PS → c_1.

PS → WPM2I: For w_{+0} and w_{-0} see the PS → PM2I analysis since $w_{+0} = t_{+0}$ and $w_{-0} = t_{-0}$. For w_{+i}, $0 < i < m$ (w_{-i} is similar):

```
(S1) for j = i until m-1 do
(S2)      e [$1^{m-j}X^j$]
(S3)      s [$X^m$]
(S4) e [$X^m$]
(S5) s [$X^m$]
(S6) for j = 2 until i do
(S7)      e [$1^{i-j}0^{m-i+1}X^{j-1}$]
(S8)      s [$X^m$]
```

PE	DTR	S2 j=1 DTR	S3 j=1 DTR	S2 j=2 DTR	S3 j=2 DTR	S4 DTR	S5 DTR
000	000	000	000	000	000	111	111
001	001	001	100	100	111	000	110
010	010	010	001	001	100	010	000
011	011	011	101	101	010	100	001
100	100	100	010	111	001	110	010
101	101	101	111	010	110	001	011
110	110	111	011	110	101	011	100
111	111	110	110	011	011	101	101

Example of PS → w_{+1}.

PS → Illiac: Follows from the PS → PM2I analysis.

Cube → PM2I: For t_{+i}, $0 \leq i < m$ (t_{-i} is similar):

```
(S1) $c_i$ [$X^m$]
(S2) for j = i+1 until m-1 do   $c_j$ [$X^{m-j}0^{j-i}x^i$]
```

PE	DTR	S1 DTR	S2 j=2 DTR
000	000	010	110
001	001	011	111
010	010	000	000
011	011	001	001
100	100	110	010
101	101	111	011
110	110	100	100
111	111	101	101

Example of Cube → t_{+1}.

Cube → PS: For the exchange use c_0. For the shuffle:

```
(S1) if ADDRESS(m-1) = ADDRESS(0) then A ← DTR [$X^m$]
                                    else $c_0$ [$X^m$]
(S2) for j = 1 to m-1 do
(S3)      if ADDRESS(j) $\neq$ ADDRESS(j-1)
                    then A ←→ DTR [$X^m$]
(S4)      $c_j$ [$X^m$]
(S5) if ADDRESS(m-1) = ADDRESS(0) then DTR ← A [$X^m$]
```

PE	DTR	S1 A	S1 DTR	S3 j=1 A	S3 j=1 DTR	S4 j=1 DTR	S3 j=2 A	S3 j=2 DTR	S4 j=2 DTR	S5 DTR
000	000	000	001	000	001	010	000	010	-	000
001	001	-	001	001	-	011	001	011	100	100
010	010	010	011	011	010	001	001	011	100	001
011	011	-	011	-	011	-	-	-	101	101
100	100	-	100	-	100	-	-	-	010	010
101	101	101	100	100	101	110	110	100	011	110
110	110	-	110	110	-	100	110	100	011	011
111	111	111	110	111	110	101	111	101	-	111

Example of Cube → Shuffle.

<u>Cube → WPM2I</u>: For w_{+0} and w_{-0} see the Cube → PM2I analysis since $w_{+0} = t_{+0}$ and $w_{-0} = t_{-0}$.

(S1) c_i $[X^m]$

(S2) <u>for</u> $j = i+1$ <u>until</u> $m-1$ <u>do</u> $c_j[X^{m-j}0^{j-i}X^i]$

(S3) c_0 $[0^{m-i}X^i]$

(S4) for $j = 1$ until $i-1$ do $c_j[0^{m-i}X^{i-j}0^j]$

PE	DTR	S1 DTR	S2 j=2 DTR	S3 DTR
000	000	010	110	111
001	001	011	111	110
010	010	000	000	000
011	011	001	001	001
100	100	110	010	010
101	101	111	011	011
110	110	100	100	100
111	111	101	101	101

Example of Cube → w_{+1}.

<u>Cube → Illiac</u>: Follows from the Cube → PM2I analysis.

<u>WPM2I → PM2I</u>: For t_{+0} use w_{+0} and for t_{-0} use w_{-0}. For t_{+i}, $0 < i < m$ (t_{-i} is similar):

(S1) $A \leftarrow DTR$ $[1^m]$

(S2) w_{+i} $[X^m]$

(S3) $B \leftarrow DTR$ $[X^m]$

(S4) $DTR \leftarrow A$ $[1^m]$

(S5) w_{-0} $[X^m]$

(S6) $B \leftarrow DTR$ $[0^{m-i}X^i]$

(S7) w_{+i} $[1^{m-1}0]$

(S8) $B \leftarrow DTR$ $[0^{m-i}1^i]$

(S9) $DTR \leftarrow B$ $[X^m]$

PE	DTR	S1 A	S2 DTR	S3 B	S4 DTR	S5 DTR	S6 B	S7 DTR	S8 B
000	000	-	111	111	111	110	110	110	110
001	001	-	110	110	110	000	000	111	111
010	010	-	000	000	000	001	000	001	000
011	011	-	001	001	001	010	001	010	001
100	100	-	010	010	010	011	010	011	010
101	101	-	011	011	011	100	011	100	011
110	110	-	100	100	100	111	100	111	100
111	111	111	101	101	111	111	101	111	101

Example of WPM2I → t_{+1}.

<u>WPM2I → PS</u>: For exchange see the WPM2I → Cube analysis, since $c_0 = e$.

For the Shuffle: Same as PM2I → PS, using w_{+i} in place of t_{+i} and w_{-i} in place of t_{-i}.

<u>WPM2I → Cube</u>: For c_i, $0 \leq i < m$:

(S1) $A \leftarrow DTR$ $[X^{m-(i+1)}1X^i]$

(S2) w_{+i} $[X^{m-(i+1)}0X^i]$

(S3) $A \longleftrightarrow DTR$ $[X^{m-(i+1)}1X^i]$

(S4) w_{-i} $[X^{m-(i+1)}1X^i]$

(S5) $DTR \leftarrow A$ $[X^{m-(i+1)}1X^i]$

PE	DTR	S1 A	S2 DTR	S3 A	S3 DTR	S4 DTR	S5 DTR
000	000	-	000	-	000	010	010
001	001	-	001	-	001	011	011
010	010	010	000	000	010	010	000
011	011	011	001	001	011	011	001
100	100	-	100	-	100	110	110
101	101	-	101	-	101	111	111
110	110	110	100	100	110	110	100
111	111	111	101	101	111	111	101

Example of WPM2I → c_1.

<u>WPM2I → Illiac</u>: Follows from the WPM2I → PM2I analysis.

<u>Illiac → PM2I</u>: For t_{+i}, $m/2 \leq i < m$ (t_{-i} is similar):

<u>for</u> $j = 1$ <u>until</u> $2^i/2^{m/2}$ <u>do</u> I_{+n} $[X^m]$

Executing I_{+n} $2^i/2^{m/2}$ times is equivalent to adding 2^i, which is equivalent to t_{+i}, $m/2 \leq i < m$. For t_{+i}, $0 \leq i < m/2$ (t_{-i} is similar):

<u>for</u> $j = 1$ <u>until</u> 2^i <u>do</u> I_{+1} $[X^m]$

Executing I_{+1} 2^i times is equivalent to adding 2^i, which is equivalent to t_{+i}, $0 \leq i < m/2$.

<u>Illiac → PS</u>: For the exchange see the Illiac → Cube analysis, since $c_0 = e$.

For the Shuffle: See Orcutt's thesis [12], section III.

Illiac → Cube: For c_{m-1} see Illiac → PM2I analysis, since $c_{m-1} = t_{+(m-1)}$.

For c_i, $m/2 \leq i \leq m-2$:

```
(S1) $A \leftarrow DTR\ [X^{m-(i+1)}1X^i]$
(S2) for $j = 1$ unitl $2^i/n$ do $I_{+n}\ [X^m]$
(S3) $A \leftrightarrow DTR\ [X^{m-(i+1)}1X^i]$
(S4) for $j = 1$ until $2^i/n$ do $I_{-n}\ [X^m]$
(S5) $DTR \leftarrow A\ [X^{m-(i+1)}1X^i]$
```

			S2			S4	
		S1	j=1	S3	S3	j=1	S5
PE	DTR	A	DTR	A	DTR	DTR	DTR
0000	0000	-	1100	-	1100	0100	0100
0001	0001	-	1101	-	1101	0101	0101
0010	0010	-	1110	-	1110	0110	0110
0011	0011	-	1111	-	1111	0111	0111
0100	0100	0100	0000	0000	0100	0100	0000
0101	0101	0101	0001	0001	0101	0101	0001
0110	0110	0110	0010	0010	0110	0110	0010
0111	0111	0111	0011	0011	0111	0111	0011
1000	1000	-	0100	-	0100	1100	1100
1001	1001	-	0101	-	0101	1101	1101
1010	1010	-	0110	-	0110	1110	1110
1011	1011	-	0111	-	0111	1111	1111
1100	1100	1100	1000	1000	1100	1100	1000
1101	1101	1101	1001	1001	1101	1101	1001
1110	1110	1110	1010	1010	1110	1110	1010
1111	1111	1111	1011	1011	1111	1111	1011

Example of Illiac → c_2, when N = 16.

For $c_{(m/2)-1}$:

```
(S1) for $j = 1$ until $n/2$ do $I_{+1}\ [X^m]$
(S2) $B \leftarrow DTR\ [X^{m/2}1X^{(m/2)-1}]$
(S3) $I_{-n}\ [X^m]$
(S4) $DTR \leftarrow B\ [X^{m/2}1X^{(m/2)-1}]$
```

		S1	S1			
		j=1	j=2	S2	S3	S4
PE	DTR	DTR	DTR	B	DTR	DTR
0000	0000	1111	1110	-	0010	0010
0001	0001	0000	1111	-	0011	0011
0010	0010	0001	0000	0000	0100	0000
0011	0011	0010	0001	0001	0101	0001
0100	0100	0011	0010	-	0110	0110
0101	0101	0100	0011	-	0111	0111
0110	0110	0101	0100	0100	1000	0100
0111	0111	0110	0101	0101	1001	0101
1000	1000	0111	0110	-	1010	1010
1001	1001	1000	0111	-	1011	1011
1010	1010	1001	1000	1000	1100	1000
1011	1011	1010	1001	1001	1101	1001
1100	1100	1011	1010	-	1110	1110
1101	1101	1100	1011	-	1111	1111
1110	1110	1101	1100	1100	0000	1100
1111	1111	1110	1101	1101	0001	1101

Example of Illiac → c_1, when N = 16

For c_i, $0 \leq i \leq (m/2)-2$, is similar to Illiac to c_i, $m/2 \leq i \leq m-2$: subsitute "1" for "n".

Illiac → WPM2I: Use the algorithm for PM2I → WPM2I, substituting I_{+1} for t_{+0}, I_{-1} for t_{-0}, and using the Illiac → PM2I algorithm for t_{+i}, $0 < i < m$.

V. Conclusions

A model of SIMD machines, designed to reflect all of the flexibility of real SIMD machines, was presented. Five different interconnection networks that have been proposed in the literature were defined in terms of the model and evaluated. The networks were analyzed in terms of the time required for each network to simulate the others. A lower time bound for each simulation was presented and an upper time bound was demonstrated by an algorithm that performed the simulation. In most cases tight bounds were found.

An SIMD machine designer must choose a set of interconnection functions, i.e., an interconnection network, to implement. It is not possible to include all of the interconnections an SIMD machine will need to perform a large variety of computations. Thus, when choosing an interconnection network, a designer must consider the ability of the network to simulate other interconnection functions.

If an SIMD machine is being designed for a specific task, then the peculiarities of that task must be considered when choosing an interconnection network to implement. If one assumes that the machine will be a general purpose one, then the results of the theorem indicate that a hybrid network consisting of the PM2I functions and the shuffle function would be quite powerful in terms of simulation ability. This hybrid would be able to simulate any network discussed here in at most 2 steps.

The methods used to prove the lower bounds and to construct the simulation algorithms can be used to analyze and compare other networks and hybrids of the networks presented here. To construct the simulation algorithms one must consider and keep track of the flow of N data words passing through N processing elements. In addition, one must determine which data may get destroyed by a data transfer that is not a bijection and save that data in such a way that it can be identified and reloaded later.

The table in Theorem 2 provides comparison information to aid the system designer in choosing a network from among those discussed. The methods presented provide tools for the designer to use to evaluate other networks.

VI. Acknowledgements

I would like to thank Professor J. D. Ullman for his help and guidance with this research. I would also like to thank L. J. Siegel for her comments.

VII. References

[1] G. H. Barnes, et. al., "The ILLIAC IV computer," IEEE Trans. Comput., Vol. C-17 (Aug., 1968), pp. 746-757.

[2] K. E. Batcher, "STARAN/RADCAP hardware architecture," Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing, pp. 147-152.

[3] K. E. Batcher, The Multi-Dimensional Access Memory in STARAN, submitted to the IEEE Trans. Comput. Special Issue on Parallel Processing; summary in the Proceedings of the 1975 Sagamore Conference on Parallel Processing, page 167.

[4] L. H. Bauer, "Implementation of data manipulating functions on the STARAN associative processor," Proceedings of the 1974 Sagamore Computer Conference on Parallel Processing, pp. 209-227.

[5] W. J. Bouknight, et. al., "The Illiac IV system," Proceedings of the IEEE, Vol. 60, No. 4 (Apr., 1972), pp. 369-388.

[6] T. Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comput., Vol. C-23 (Mar., 1974), pp. 309-318.

[7] T. Feng, Parallel Processing Characteristics and Implementation of Data Manipulating Functions, Dept. of Electrical and Computer Engineering, Syracuse University, RADC-TR-73 189 (July, 1973).

[8] M. J. Flynn, "Very high-speed computing systems," Proceedings of the IEEE, Vol. 54, No. 12 (Dec., 1966), pp. 1901-1909.

[9] S. W. Golomb, "Permutations by cutting and shuffling," SIAM Review, Vol. 3, No. 4 (Oct., 1961), pp. 293-297.

[10] P. B. Johnson, "Congruences and card shuffling," American Mathematical Monthly, Vol. 63 (Dec., 1956), pp. 718-719.

[11] D. E. Lawrie, Memory-Processor Connection Networks, Dept. of Computer Science, University of Illinois, Rep. 557, (Feb., 1973).

[12] S. E. Orcutt, Computer Organization and Algorithms for Very-High Speed Computation, Dept. of Computer Science, Stanford University, Ph.D. Thesis, (Sept., 1974).

[13] D. Rahmlow, "Parasim," Princeton University, unpublished paper, (1974).

[14] H. J. Siegel, "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks," Proceedings of the 1975 Sagamore Conference on Parallel Processing, pp. 106-109.

[15] H. J. Siegel, SIMD Machine Interconnection Network Design, Princeton University, Department of Electrical Engineering, Computer Science Laboratory Technical Report 198, (Jan., 1976).

[16] H. J. Siegel, Single Instruction Stream-Multiple Data Stream Machine Interconnection Network Universality, Princeton University, Department of Electrical Engineering, Computer Science Laboratory Technical Report (Aug., 1976).

[17] D. L. Slotnick, et. al., "The SOLOMON computer," 1962 Fall Joint Computer Conf., AFIPS Proc., Vol. 22 (1962), pp. 97-107.

[18] H. S. Stone, "Parallel processing with the perfect shuffle," IEEE Trans. Comput., Vol. C-20 (Feb., 1971), pp. 153-161.

[19] R. C. Swanson, "Interconnections for parallel memories to unscramble p-ordered vectors," IEEE Trans. Comput., Vol. C-23, No. 11 (Nov., 1974), pp. 1105-1115.

[20] D. E. Wilson, "The PEPE support software system," Sixth Annual IEEE Computer Society International Conference (1972), pp. 61-64.

EFFECTIVENESS OF SOME PROCESSOR/MEMORY INTERCONNECTIONS

K.Y. Wen and D.H. Lawrie
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract -- This paper is an overview of some of our efforts to determine the combined effectiveness of program restructuring techniques and various processor/memory interconnection networks from the user's point of view. This paper first attempts to investigate the structural and functional similarities and differences of some of these networks. Some new results on network properties and their capabilities in handling computations are also presented. Then the paper describes how we can apply the theoretical results of various networks to predict their performances in a real program environment, which is the true measure of network effectiveness. The result of this study enables us to answer some long standing questions about the real effectiveness of various interconnection schemes.

Introduction

Recently, component speeds have continued to improve. However, there are certain physical limitations to component speeds. Multiprocessing then seems to be an area to show the most promise for any further speedup of computations. The arrival of the cheap but powerful LSI microprocessors greatly increases the attractiveness of multiprocessing systems. However, a big problem arises in finding the best way to interconnect all the processors. The questions that are yet to be answered are what kind of network should we use, how should we compile or restructure computation algorithms in order to use it, and how well does it work on ordinary application programs.

Many interconnection schemes have been proposed or built in recent years. Thurber [1] gives a survey on some of the more important ones. However, each of the networks proposed or built has different requirements to fulfill and their implementations are based on different theoretical backgrounds. Frequently, their capabilities are incompletely known, and their control algorithms are poorly understood. Hence it is very difficult to categorize or assess the merits of each of these networks.

This paper first summarizes some new results on certain network properties and their capabilities in handling computations. Then the paper describes how we can apply the theoretical capabilities of various networks to predict their performances in a real program environment, which is the true measure of network effectiveness. This performance prediction is being done using an Analyzer/Simulator program, which can be used as a tool to compare various parallel architectures.

In general, processor(/memory) interconnection networks can be divided into two classes. The first class has multiple stages of switching elements. The second class has only a single stage of switching elements and this stage may have to be recycled many times to obtain certain permutations. Examples of the first class are the Batcher network[2], the Benes network[3] the omega network[4], the barrel shifter, and the Feng's data manipulator[5]. Networks such as the Illiac IV connection[6], the Swanson connection[7], the ± 1 shift network and the perfect shuffle network[8] are good examples of the one stage networks. Although single stage networks may be slower in performing general permutations of data than the multistage networks, they are much cheaper in comparison. If we can restructure and recompile some of the commonly used computation algorithms into algorithms which fully utilize the available connectivities, we can retain the performance level while drastically decreasing the cost of the processing system.

One of the multiple-stage networks which has been of particular interest to us in recent years is the omega network. This network cannot perform all connections of its inputs to outputs, yet it is capable of producing most of the connections required by numerical programs. Because of the incomplete capabilities of this network, it is necessary to analyze the network to determine exactly which connections it can produce. A number of these connections have been demonstrated in [4]. Before proceeding with a discussion of the program analyzer and simulation experiments, we summarize in the next two sections some new theorems about omega networks which demonstrate connection capabilities that are important for handling certain numerical algorithms found in many application programs.

Omega Partition Theorems

One important property of the omega network is its ability to be partitioned. The theorems in this section will show that a large omega network can be regarded as a conglomeration of many smaller omega networks, each passing a different smaller omega-passable connection function. These partition theorems help to establish many capabilities of a larger size network on smaller, partitioned connections.

*This work was supported in part by NSF Grant MCS73-07980-A03.

Example:

Given an 8x8 omega network. Assume that source ports 0-3 want to do an end-around 1-shift. Assume also that destination ports 4-7 request data from port 5. So the complete set of source destination pairs is P={(0,1),(1,2),(2,3),(3,0),(5,4),(5,5),(5,6),(5,7)}. We know that a 4x4 omega network can perform an end-around 1-shift, as well as a one-to-many broadcasting function. By using the partition theorem stated below, we can be sure that an 8x8 omega network can pass P.

Definition 1

Let $P_L=\{(s_i,d_i)\,|\,0\le i<L\}$ and $P_{M_i}=\{(t_{ij},e_{ij})\,|\,0\le j<M\}, 0\le i<L$.

We define $P_N=P_L x\{P_{M_0},P_{M_1},\ldots\ldots P_{M_{L-1}}\}$

$$=\{(s_iM+t_{ij},d_iM+e_{ij})\,|\,0\le i<L, 0\le j<M\}$$

For example, let L=4, M=4 and N=16. If

$P_{M_0}=\{(0,1),(1,2),(2,3),(3,0)\}$,

a 1-shift permutation,

$P_{M_1}=\{(0,0),(0,1),(0,2),(0,3)\}$,

a 1-to-4 broadcast connection,

$P_{M_2}=\{(0,3),(1,2),(2,1),(3,0)\}$,

a flip permutation,

$P_{M_3}=\{(0,0),(1,3),(2,2),(3,1)\}$,

a 3-order unscrambling, and

$P_L=\{(0,2),(1,3),(2,0),(3,1)\}$,

a 2-shift permutation.

Then by definition, $P_N=\{(0,9),(1,10),(2,11),(3,8),(4,12),(4,13),(4,14),(4,15),(8,3),(9,2),(10,1),(11,0),(12,4),(13,7),(14,6),(15,5)\}$.

In words, the sources and destinations of P_N are divided into 4 partitions. P_L is the inter-partition permutation function, and P_{M_i}'s are the individual partition permutations. P_L moves partition #0 to partition #2 and then the individual elements in partition #2 will be moved according to P_{M_0}, and so on. A pictorial illustration of P_N is shown in Figure 1.

Lemma 1

(Equivalent Statement of Theorem 2 in [4])

Given a set of desired input-output connection $P_N=\{(s_i,d_i)\,|\,0\le i<N\}$, then NxN omega network passes P_N if and only if for all s-d pairs in P_N and for all $m=2^k$, where $1\le k\le \log N$, $s_i \underset{N}{\equiv} s_j$, or $s_i \underset{m}{\not\equiv} s_j$ or $d_i \overset{m}{\underset{N}{\not\equiv}} d_j$.

Alternatively speaking, a binary omega does not pass a connection P if and only if there exist (s_i,d_i) and (s_j,d_j) and where k=logm, such that:

1) the leading (logN-k) bits of s_i and s_j are not equal $(s_i \overset{m}{\underset{N}{\not\equiv}} s_j)$,

2) the trailing k bits of s_i and s_j are equal $(s_i \underset{m}{\equiv} s_j)$, and

3) the leading (logN-k) bits of d_i and d_j are equal $(d_i \overset{m}{\underset{N}{\equiv}} d_j)$.

Theorem 1

Let $\Omega_L\uparrow P_L$ and $\Omega_M\uparrow P_{M_i}, 0\le i<L$, and let N=LxM, then $\Omega_N\uparrow P_N=P_L \times \{P_{M_0},P_{M_1},\ldots\ldots P_{M_{L-1}}\}$.

Proof

Assume $\Omega_N\not\uparrow P_N$. By Lemma 1, there exist $S_1=s_pM+t_{pq}$, $D_1=d_pM+e_{pq}$, $S_2=s_uM+t_{uv}$, $D_2=d_uM+e_{uv}$ and X such that $S_1 \underset{N}{\not\equiv} S_2$, $S_1 \underset{X}{\equiv} S_2$ and $D_1 \overset{X}{\underset{N}{\equiv}} D_2$.

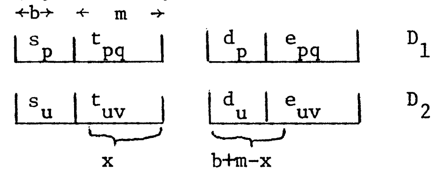
Let m=logM, n=logN, b = logL, and x = logX.

If $X\ge M$, pictorially we have:

$$\begin{array}{lccccl} & \leftarrow b\rightarrow & \leftarrow m\rightarrow & & & \\ S_1 & \boxed{s_p} & \boxed{t_{pq}} & \boxed{d_p} & \boxed{e_{pq}} & D_1 \\ S_2 & \boxed{s_u} & \boxed{t_{uv}} & \boxed{d_u} & \boxed{e_{uv}} & D_2 \\ & & \underbrace{\qquad}_{x} & \underbrace{\quad}_{b+m-x} & & \end{array}$$

Here the trailing x bits of S_1 and S_2 are equal, but the leading (b+m-x) bits are not equal, and the leading (b+m-x) bits of D_1 and D_2 are equal. Since $x \leq m$, the trailing $(a=x-m)$ bits of s_p and s_u are equal but the leading $(b-a)$ bits are different, and the leading $(b-a)$ bits of d_p and d_u are equal. This contradicts $\Omega_L \uparrow P_L$.

If X<M, pictorially we have:



Since the leading (b+m-x) bits of $\underset{X}{D}_1$ and D_2 are equal and m>x, we have $d_p=d_u$ and $e_{pq} \equiv_M e_{uv}$.

Since $\Omega_L \uparrow P_L$ and $d_p=d_u$, p has to be equal to u. So $s_p=s_u$. This implies that $t_{pq} \neq t_{uv}$ since $S_1 \neq_N S_2$. $S_1 \equiv_X S_2$ and $X<\underset{X}{M}$ implies that $t_{pq} \equiv_X t_{uv}$. Setting p=u, we get $e_{pq} \equiv_M e_{pv}$, $t_{pq} \neq_M t_{pv}$ and $t_{pq} \equiv_X t_{pv}$. They imply that $\Omega_M \not\uparrow P_{M_p}$, which is a contradiction. Hence, Theorem 1 is proved.

In Theorem 1, the tag bits denoting the partitioning are the most significant log L bits. In the following two theorems, we extend the result to any set of log L bits in the tag representation.

Let us look at (s_i,d_i) and (s_j,d_j) like the following:

$$(s_i,d_i): (\underline{x}_1 x_2 x_3 \underline{x}_4 x_5 x_6 x_7 \underline{x}_8 x_9 \underline{x}_{10}, \underline{y}_1 y_2 y_3 \underline{y}_4 y_5 y_6 y_7 \underline{y}_8 y_9 \underline{y}_{10})$$

$$(s_j,d_j): (\underline{a}_1 a_2 a_3 \underline{a}_4 a_5 a_6 a_7 \underline{a}_8 a_9 \underline{a}_{10}, \underline{b}_1 b_2 b_3 \underline{b}_4 b_5 b_6 b_7 \underline{b}_8 b_9 \underline{b}_{10})$$

Assume there are log L underlined bits and log M non-underlined bits.

<u>Theorem 2</u>

Assume all the underlined bits of (s,d) satisfy an omega passable connection[†] P_L, and all the non-underlined bits of (s,d) satisfy an omega passable <u>connection</u> P_{M_k}, where k represents the total numerical value of the underlined bits of $\underline{s}$. Then {(s,d)} is passable by an omega network of size LMxLM.

Proof: see [9].

[†]Note that a connection can be a broadcasting function, while a permutation cannot, i.e. a connection can be one-to-many while a permutation must be one-to-one.

<u>Theorem 3</u>

Assume all the underlined bits of (s,d) satisfy an omega passable <u>permutation</u> P_L, and all the non-underlined bits of (s,d) satisfy an omega passable <u>connection</u> P_{M_k}, where k represents the total numerical value of the underlined bits of $\underline{d}$. Then {(s,d)} is passable by an omega network of size LMxLM.

Proof: see [9].

It should be noted that all three theorems allow P_{M_k}'s to be any connection function, and P_L can be any connection in Theorems 1 and 2 but must be a <u>permutation</u> in Theorem 3.

This partitioning property of the omega network proves to be vital for the efficient handling of many algorithms, especially the Recurrence Solvers, as discussed later in this paper.

Another important property of the omega network is its ability to produce broadcast connections, i.e. one-to-many mappings of inputs to outputs. This ability is necessary for example in certain matrix multiplication algorithms and algorithms for solving recurrence systems. We will summarize the broadcast theorems in the next section.

<u>Omega Broadcast Theorems</u>

Theorems 10 and 11 of [4] describe certain broadcasting functions for small square submatrices of data. In this section, we are going to extend these results to 3-dimensional arrays, not necessarily of equal size edges.

We use the notation (k,x,y) <a,b,c> to denote element (k,x,y) of an axbxc array. Here $0 \leq k<a$, $0 \leq x<b$, $0 \leq y<c$. Also (k,x,y) <a,b,c> $\longrightarrow$ (*,x,y) <a,b,c> symbolizes the mapping of the element (k,x,y) to elements (0,x,y), (1,x,y),....(a-1,x,y).

Now we can show six extensions of the broadcast theorems.

For constant k and for all values of x and y:

1) $\Omega_{abc} \uparrow$ {(k,x,y) <a,b,c> $\longrightarrow$ (*,x,y) <a,b,c>}
2) $\Omega_{abc} \uparrow$ {(k,x,y) <a,b,c> $\longrightarrow$ (x,*,y) <b,a,c>}
3) $\Omega_{abc} \uparrow$ {(k,x,y) <a,b,c> $\longrightarrow$ (x,y,*) <b,c,a>}
4) $\Omega_{abc} \uparrow$ {(k,x,y) <a,b,c> $\longrightarrow$ (y,x,*) <c,b,a>} iff $a \geq c$
5) $\Omega_{abc} \not\uparrow$ {(k,x,y) <a,b,c> $\longrightarrow$ (*,y,x) <a,c,b>}
6) $\Omega_{abc} \not\uparrow$ {(k,x,y) <a,b,c> $\longrightarrow$ (y,*,x) <c,a,b>}

These broadcast theorems are also essential in implementing the recurrence algorithms discussed in Section 4 and are some of the more important properties of the omega network. They will not be proved here. However, proofs can be found in [9].

Psuedo Compilation of Code for Simulation of Parallel Architecture

In order to evaluate the true effectiveness of a parallel architecture, we must hypothesize a compiler capable of compiling ordinary programs into code which most effectively utilizes the architecture, especially the data alignment capabilities. The resulting code could then be simulated and the important performance measures determined. This is one of the goals of an ongoing effort at the University of Illinois. A Program Analyzer has been implemented which accepts Fortran source programs, and by detailed analysis of the control and data dependencies it produces a highly parallelized version of the original program (see [10]). Next, this parallelized version is input to another program, the Resource Request Generator (RRG), which attempts to compile the parallelized program into simulatable code. This pseudo compilation is done based on the capabilities of the architecture to be studied, including the type of interconnection network. Finally, the output of the RRG is input to a simulator capable of simulating a wide variety of architectures. The Program Analyzer is described elsewhere [10] and we will not discuss it here. In this section we will briefly describe the RRG.

The most easily recognizable form of parallelism is typified by a matrix addition shown below.

```
      DO 10 I=1,N

      DO 10 J=1,M

   10 A(I,J)=B(I,J)+C(I,J)
```

The Program Analyzer will determine that there are no restrictions on how this computation can be 'sliced'. The addition can proceed by rows, by column, or in fact, the elements of the resultant matrix can be computed in virtually any order. The task of the RRG is to decide on the best way to slice this computation based on the size of the matrices, the number of available processors, the matrix storage scheme, and the type of alignment network. Various aspects of this problem are discussed in the literature. For example, Budnik and Kuck [11] and Lawrie [4] discussed ways of organizing the memories to allow conflict-free access to various slices of arrays. Linear skewing is a standard technique. However, the data output will sometimes form a p-ordered vector, which cannot be unscrambled by means of a simple shifter. Lawrie [4] discussed the alignment requirements for some of the most common types of array access.

The parallelism in other computations may not be so obvious. Consider for example the computation shown below.

```
      DO 10 I=1,N

   10 X(I)=A(I)*X(I-1) +B(I)
```

This is an example of what we call a recurrence computation. Special techniques must be used to perform such a computation on a parallel processor in order to maintain reasonable performance. Kogge and Stone [12], Heller [13] and Chen and Kuck [14] have shown various algorithms which can be used to speed up such computations. However, we will not discuss all of these techniques here. Suffice to say that the Program Analyzer detects such computations, and offers the RRG a variety of options for their solutions.

The adaptation of a computation onto a parallel processor must be tailored according to the limited number of available connections of the alignment network to minimize alignment time. In the extreme cases, the alignment network may have only a limited number of connections (like the Illiac IV shifter or a one-stage perfect shuffle network). To obtain any general permutation, the network has to be recycled many times. For example, a one-stage perfect shuffle network may require $0(\sqrt{N})$ alignment steps before we can start on a processing step. By carefully rearranging some of the operation sequences in normal algorithms and by assigning intermediate storage patterns in a deliberate fashion, we can sometimes reduce the number of alignment operations per processing step down to a constant (not dependent on N).

A good example is matrix multiplication. A Fortran code section that performs matrix multiplication is as follows:

```
      DO 10 I=1,N

      DO 10 J=1,N

      DO 10 K=1,N

   10 A(I,J)=A(I,J)+B(I,K) *C(K,J)
```

Notice the obvious parallelism in the I and J indices, but that the K index involves a recurrence. Assuming the number of processors is small compared to N^2, an efficient way to perform the calculation would be to compile the product by columns (parallel on I), or by rows (parallel on J) as shown below.

```
      DO 10 I=1,N

      DO 10 K=1,N

   10 A(I,*)=A(I,*)+B(I,K)*C(K,*)
```

This algorithm will require $0(N^2)$ shifts to align the operand matrices. A one-stage perfect shuffle network simulating an omega network will take log N steps per shift, and the Illiac IV type of switch will take $O(\sqrt{N})$ steps per shift on the average. So a total of $O(N^2 \log N)$ or $O(N^2\sqrt{N})$ routing steps are required for matrix multiplication. However, using Algorithm 1 which follows, we need only $O(N^2)$ steps. This algorithm can use either a one-stage perfect shuffle network or an Illiac IV type of switch.

Assume we want to multiply two matrices B and C to form A and that they are all of size NxN. The first method uses N processors and requires that the storage scheme for the matrices be 1-skew and 1-skip. The storage pattern is shown in Figure 2. Each processor will have a corresponding memory from which it can fetch data. Any data a processor wants but not in its own memory will have to be routed from the other processors. This algorithm also calculates the relative address (RA) for each array it references.

Each processor has a wired-in processor port number, PPN ($0 \leq PPN < N-1$). T is a temporary array.

Algorithm 1

```
for IC = 0 to N-1 do
        fetch B(RA=IC) into R1
        IR <-- (PPN-IC) mod N
    for IT = 0 to N-1 do
        fetch C(RA = IR) into R2
        R3 <-- R1*R2
        G-permute IR
        store T(RA=(PPN-IR)mod N) from R3
        G-permute R1
    end
    R1 <-- 0
    for IT = 0 to N-1 do
        fetch T(RA=IR) into R2
        R1 <-- R1 + R2
        G-permute R1
        G-permute IR
    end
    store A (RA=IC) from R1
end
```

Algorithm 1 depends on the ability of the alignment network to do a 'G-permutation'. Definition: A G-permutation is defined as a permutation G such that $G, G^2, G^3, \ldots G^{N}$† are distinct and form a group with $G^N = I$, the identity permutation.

Every G permutation can be uniquely represented as a cycle $(i_o, i_1, \ldots i_{N-1})$ where $G(i_o) = i_1$, $G(i_1) = i_2, \ldots G(i_{N-1}) = i_o$.

Two obvious G-permutations are the +1 shift permutation and the -1 shift permutation. In general, +k shift and -k shift permutations will be G-permutations if k is relatively prime to N. Some nonshifting G-permutations can be found using a perfect shuffle based permutation. The G-permutations have a general form of:

$$G(i) = [2i+b(i)] \bmod N$$
$$\text{where } b(i) = b(i+N/2) \ \forall\ i=0\ldots N/2-1$$
$$\text{and } b(i) = 0 \text{ or } 1\ \forall\ i$$

A list of all $\{b(i), i=0\ldots.N/2-1\}$ that will give G-permutations for N=4 and 8 and the corresponding G-permutations are listed in Table 1.

Size	b(i)	G-permutation
4	1 1	(0 1 3 2)
8	1 1 0 1	(0 1 3 7 6 5 2 4)
	1 0 1 1	(0 1 2 5 3 7 6 4)

Table 1

The significance of this result is that for certain one stage networks, if there exists a G-permutation, then each intermediate routing will take only $O(1)$ time instead of $O(\log N)$ time or $O(\sqrt{N})$ time. This greatly reduces the alignment time for the system.

The RRG uses results like these to produce code which effectively utilizes a given architecture.

Experiments and Preliminary Results

Over the next six to twelve months we will be conducting experiments to determine the real effectiveness of various array processor architecturals. In this section we will describe the design of these experiments and present some preliminary results.

The simulator can simulate essentially any architecture. Possibilities include systems composed of parallel processing elements, pipelined processors, or combinations of both, and various alignment networks and memory systems. The architecture may include one or more scalar processors, control units, and I/O subsystems. The simulation can proceed at various levels of detail, from a gross level where a group of processing elements forms a single system resource, to finer levels where even register usage is accounted for.

The output of the simulator is a set of performance measures. One such measure is T_p, the time required for simulated execution of the program using p processors. Another measure is the speed factor, F_p, which is defined as T_1/T_p. In addition, the simulator calculates measures of the utilizations of various system resources. If the architecture is a SIMD multiprocessor, i.e. an

†G^i implies i consecutive applications of the permutation G to the input set.

array of processors, then the processor utilization is broken down into several separate utilizations, U_a, U_s, and U_{IF}. First, U_a, the array duty cycle, is the percentage of time that at least one processor is performing a computation. However, whenever an array operation is being performed, only some of the processors may be actually doing useful work. This is measured by the slicing utilization, U_s. For example, to add two 30 element vectors together using 20 processors would require two steps. The first step would form the first 20 sums and would use all 20 processors resulting in a slicing utilization, U_s, of 100%. The second step would form the last 10 sums using only 10 processors and would result in U_s=50%. The overall U_s would then be 75%. Finally, some processors are turned off because of IF statements in the original programs, and this is measured by U_{IF}. For example, assume that in the following program, 1/3 of the B(I) are less than zero:

```
      DO 10 I=1,30
   10 IF (B(I).GE.0) A(I)=A(I)+B(I)
```

Then U_{IF}=67%. Thus, using 20 processors on this program, U_a might be 80%, e.g. because the processors are waiting for memory access or data alignment. Of this 80% of the time, only 75% of the processors could be used because of the difference between the number of processors and the array size (U_s=75%), and of these 75% of the processors, only 67% are turned on (U_{IF}=67%). Thus, the total average processor utilization, U_T, is equal to $U_a*U_s*U_{IF}$ = 80%*75%*67% = 40%. By separating the components of processor utilization in this way we can determine the source of processor inefficiencies.

Our initial experiments will deal with the effects of the following architectural parameters:

1) The number of array processors, and the speed of the processors relative to the array memory system. Initially, the processors will be restricted to a single group of processors operating from a single instruction stream (SIMD).

2) The presence or absence of an independent scalar processor and/or memory. The absence of a scalar processor forces scalar operations to be performed by the array processors.

3) The memory system, including the array storage scheme (1-skew, etc.), and the number of memories(power of two or prime).

4) The type of alignment network:
   a) crossbar
   b) omega network
   c) $\pm 1, \pm\sqrt{p}$ shifter (Illiac IV)

These parameters will be studied for a large variety of application programs, and in addition the size of the application programs (i.e. the array sizes) will be varied in order to produce families of performance figures.

The tables below present some preliminary reresults of experiments on three programs. We would like to stress at this point that these results are preliminary. The three programs can hardly be construed as representative of any large population of applications. The first program, ADVV, is a 4-point relaxation scheme. ADVV was chosen because of its highly parallel nature. The second program, ELMBAK, forms the eigenvectors of a real matrix by back transforming those of the corresponding upper Hessenberg matrix. ELMBAK is reasonably complicated, but has no recurrences. The third program, SLEQ1, is a Gauss-Jordan reduction program. SLEQ1 was chosen because it contains a representative recurrence relation. We present the results of these three programs only as an indication of the types of results we expect from our experiments.

Table 2 shows the speed factor, $F_p=T_1/T_p$, and processor utilization U_T using 16 processors, 17 memories, a crossbar alignment network, skewed storage, and separate scalar processor and memory. The results are presented as a function of N, the data array sizes. Notice for ADVV the speed factor quickly approaches the maximum value of 16. Processor utilization ranges from 43% to 71%. The result for N=16 indicates that U_a = 70% (U_s=100% since N=p=16 and for ADVV, U_{IF}=100%). Thus, the processors are only busy 70% of the time due to non-perfect overlap of array processor operations with alignment, memory, and scalar operations. However, the speed factor is 16 which would indicate a similar degree of non-perfect overlap in a comparable serial processor. The other programs ELMBAK and SLEQ1 indicate much lower speed factors and utilizations. SLEQ1 contains recurrences, which are handled in parallel but much less efficiently than the pure vector operations in ADVV. Notice, however, that even though SLEQ1 contains a recurrence, the speed factor of 14.5 is very close to the maximum of 16 when N is 60. We believe it is significant that we are able to handle recurrences this well.

The reason the ELMBAK results are so low illustrates an interesting situation. At the present time programs are compiled into three address vector or scalar instructions. If the vectors are of sufficient length, then an implicit loop is established in order to cycle the processors, memories, etc. a sufficient number of times. Within this implicit loop there is usually overlap between processor, alignment and memory operations. However, between separate vector instructions, there is no overlap. Thus, one instruction must finish before the next starts. This is what causes the low

Program	N=10	N=16	N=40	N=60	N=100
ADVV	9.7, 43%	16.0, 70%	14.1, 62%	15.9, 70%	16.2, 71%
ELMBAK	2.0, 6%	3.0, 10%	5.9, 23%	8.0, 32%	9.6, 39%
SLEQ1	4.5, 14%	6.8, 21%	9.6, 30%	14.5, 45%	--

Table 2. Speed (F_{16}) and processor utilization (U_T) using 16 processors, 17 memories (cf [11]), crossbar alignment network and skewed storage. N is the data array size .

	Crossbar		Omega		Illiac IV	
Program	Straight	Skewed	Straight	Skewed	Straight	Skewed
ADVV	1416 (9.7)	1416 (9.7)	1416 (9.7)	1416 (9.7)	1384 (9.9)	1384 (9.9)
ELMBAK	1760 (2.0)	1760 (2.0)	1760 (2.0)	1760 (2.0)	1261 (2.8)	1282 (2.8)
SLEQ1	2636 (4.5)	2636 (4.5)	2636 (4.5)	2636 (4.5)	*	*

*SLEQ1 contains recurrences which we have not yet programmed on Illiac type interconnections.

Table 3. Execution time, T_p, and speed factor (F_p) using various alignment networks and skewing schemes. (p=16).

figures for ELMBAK. This indicates to us that it is important to design the vector instructions and control unit so that different vector instructions overlap each other.

It is also interesting to note that F_p and U_T continue to increase with N for both ELMBAK and SLEQ1. This is due to increased overlap of operations within the implied loops of vector instructions and, in SLEQ1, more efficient recurrence algorithms which are used when N is sufficiently larger than the number of processors.

Table 3 indicates the effectiveness of various alignment networks and skewing schemes. As we can see, the crossbar and omega networks performed equally well. The Illiac network performed somewhat better, at least for ADVV and ELMBAK. This is due to two facts. First, the Illiac network was set to operate four times faster than the other networks. This reflects the difference in the complexity of the networks. Second, we were able to "compile" the programs using very simple alignment requirements which could be easily handled by all three networks. The lack of difference between straight storage and skewed storage is also a reflection of this second point. We were able to compile the programs so they only needed access to rows, and thus they do not benefit from skewed storage. However, we do not believe this result will hold for larger, more complicated programs.

Table 4 illustrates another interesting result. One question which continually plagues machine designers concerns the relative speed of the memory and processor. Should the memory be the same speed as the processor, twice as fast, or three times as fast? The answer depends on many things: the design of the machine instructions, the size of arithmetic expressions in the source program, etc. Table 4 shows the execution time, T_p, and processor utilization U_T for three different cases. In column 1, the processor array, alignment network, and memory all have the same cycle time. In column 2, the alignment network alone has been made twice as fast. There is very little difference between columns 1 and 2. This is because the faster crossbar switch is only effective when data alignments are required in the absence of memory accesses. None of these three programs required such alignment. The small difference present between columns 1 and 2 simply represents a shorter overall time for a "short" vector operation in the absence of inter-instruction overlap.

Column 3 of Table 4 corresponds to a machine whose alignment network and memories are twice as fast as the processor array. For ADVV, the improvement in T_p is noticeable but not significant. This is because ADVV has relatively large expressions in the source program so the ratio of memory to processor operations is close to 1:1. Thus ADVV does not need a very fast memory. For ELMBAK and SLEQ1 however, the improvement in T_p is more significant. This would indicate that, at least for these programs, the faster memory might be cost effective.

Table 5 shows the effectiveness of an independent scalar processor and scalar memory on T_p. Also included in the table are the utilizations of scalar memory and scalar processor respectively. A scalar processor and memory should be effective for several reasons. First, without a scalar memory, when a scalar is being broadcast over all elements of an array, the scalar operand would have to be fetched from the array memory and aligned (broadcast). This constitutes wasteful use of the array memory. Second, the use of both scalar memory and processor would allow some scalar operation to be done simultaneously with array operations. Thus we would be able to overlap or mask out certain truckulent serial operations in the program.

In Table 5 we can see that the scalar processor causes no improvement in T_p and the scalar memory results in only marginal improvement, even though both are utilized to some extent. However, we believe that our "compiler" can be improved so as to utilize the scalar hardware more effectively. This will involve improving the inter-instruction overlap and more accurate accounting for such things as subscript calculation.

Conclusion

In this paper we have presented an overview of our efforts to evaluate high speed computer architectures in an environment of real programs. This has involved a number of separate efforts. First, a Program Analyzer described elsewhere (c.f. [10]) was developed which can analyze a program and allow us to restructure the program so it can more effectively utilize a given architecture. Second, we investigated the capabilities of various alignment networks and designed algorithms which could utilize these capabilities. Finally, we developed a simulator to test our theories. We believe that by using these tools we will be able to answer some long standing questions about the design and effectiveness of high speed computers.

Acknowledgement

The Program Analyzer is the result of the efforts of Bruce Leasure, Ross Towle, Mike Wolfe and Mike Wilde. Donald Chang wrote the simulator. Yuzo Hayashi provided assistance in experimental design and debugging. Much progress in the solution of recurrences has been provided by S.C. Chen and Ahmed Sameh. We are deeply indebted to these colleagues, and especially to Dave Kuck who has stimulated and overseen the entire effort.

References

[1] K. J. Thurber, "Interconnection Networks--A Survey and Assessment," AFIPS Conference Proceedings, Vol. 43, pp. 909-919, May 1974.

[2] K. E. Batcher, "Sorting Networks and Their Applications," Proceedings of the 1968 SJCC, pp. 307-314.

[3] V. E. Benes, Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, New York, 1965.

[4] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Transactions on Computers, pp. 1145-1155, December 1975.

[5] T. Feng, "Data Manipulating Functions in Parallel Processors and Their Implementations," IEEE Transactions on Computers, pp. 309-318, March 1974.

[6] G. H. Barnes, et al, "The Illiac IV Computer," IEEE Transactions on Computers, pp. 746-757, August 1968.

[7] R. C. Swanson, "Interconnections for Parallel Memories to Unscramble p-ordered Vectors," IEEE Transactions on Computers, pp. 1105-1115, November 1974.

[8] H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Transactions on Computers, pp. 153-161, February 1971.

[9] K. Y. Wen, "Interprocessor Connection--Capabilities, Exploitation, and Effectiveness," Ph.D. Thesis Dissertation, August, 1976.

[10] B. R. Leasure, "Compiling Serial Languages for Parallel Machines," M.S. Thesis, University of Illinois, 1976.

[11] P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," IEEE Transactions on Computers, pp. 1566-1569, December 1971.

[12] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," IEEE Transactions on Computers, pp. 786-792, August 1973.

[13] D. Heller, "On the Efficient Computation of Recurrence Relations," Inst. Comput. Appl. Sci. Eng.(ICASE), June 1974.

[14] S. C. Chen and D. J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems," IEEE Transactions on Computers, pp. 701-717, July 1975.

Program	Neither	Scalar memory only	Scalar processor only	Both scalar processor and memory
ADVV	1544 -, -	1416 12%, -	1544 -, 3%	1416 12%, 3%
ELMBAK	1854 -, -	1760 12%, -	1854 -, 12%	1760 12%, 12%
SLEQ 1	2752 -, -	2636 8%, -	2752 -, 9%	2636 8%, 9%

Table 5. The effect on execution time, T_p, of a scalar memory and scalar processor. The percentage figures are scalar memory utilization and scalar processor utilization respectively.

Program	Col 1	Col 2	Col 3
ADVV	1416 43%	1384 44%	1036 58%
ELMBAK	1760 6%	1650 7%	1023 10%
SLEQ1	2636 14%	2582 14%	1602 23%

Table 4. The effects of the relative differences in memory, alignment, and processor cycle times on T_p and U_T. (16 processors, crossbar switch, N=10, and skewed storage.)

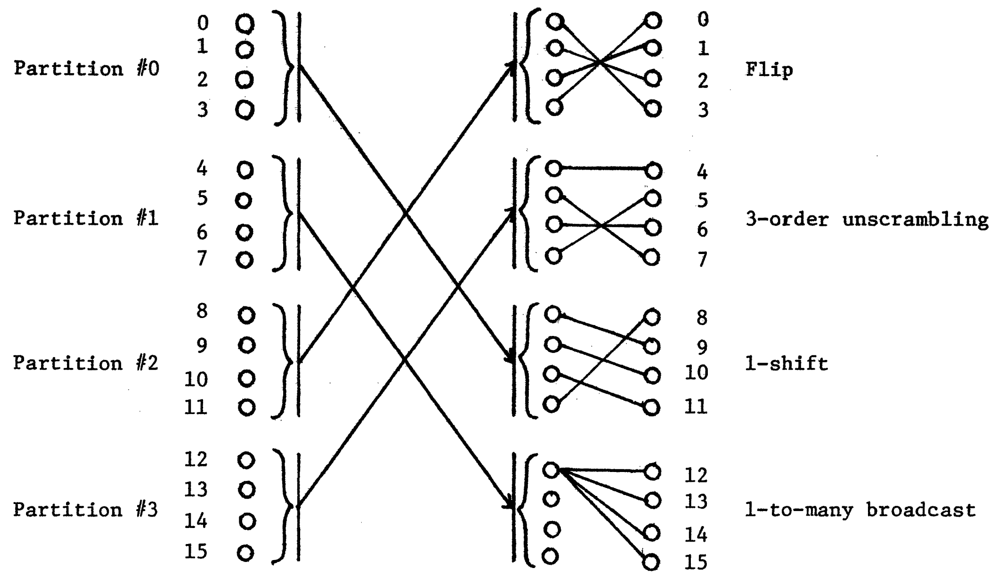


Figure 1 A Partitioned Permutation

Memory

0	1	2	3
(0,0)	(0,1)	(0,2)	(0,3)
(1,3)	(1,0)	(1,1)	(1,2)
(2,2)	(2,3)	(2,0)	(2,1)
(3,1)	(3,2)	(3,3)	(3,0)

Figure 2 1-skew 1-skip Storage Scheme

LAU SYSTEM ARCHITECTURE : A PARALLEL DATA-DRIVEN PROCESSOR BASED ON SINGLE ASSIGNMENT

by

A. Plas, et al[*], Université de Toulouse, France

Abstract -- This paper presents the architecture of a data driven processor interpreting a high level machine language.
This language is based on single assignment which allows a natural description of maximal parallelism in the program. The basic control mechanisms of a data directed execution are briefly described. The way they are implemented is stressed, principally for the control unit which replaces the program counter and related controls of a conventional Von Neumann machine.
Finally, simulation results, taken from a set of programs run on the LAU system are given and evaluated.

Introduction

This paper deals with a parallel computer architecture based on data directed execution and single assignment language. Single assignment naturally implies a maximal parallelism description of problems. First, a machine language is described, and some general aspects on the global parallel architecture are given. The following sections of the paper present the outlines of an execution processor composed of three main units, each of them being described to understand the basic hardware mechanisms replacing a conventional control unit and allowing a fully parallel, data driven interpretation of instructions.
A compiler of a high level language and a simulator have been designed. The last section presents the simulation results and gives some performances compared to sequential executions for sample programs.

Basic software features

The processor described in this paper, interprets a single assignment machine language. The object code is produced by a compiler from a high level single assignment language : LAU, [1], [2]. The single assignment rule states that an object (we prefer "object" than "variable" to denote the data entities defined by the programmer) may be assigned at most once during the "program life" [3],[4],[5],[6],[7]. Any assignment statement can be represented by :

$\emptyset = f(I)$	f : operation code
	I = (I1,I2 ... In) : input set
	$\emptyset$ = (O1,O2 ... Or) : output set

(*) Research supported under contract SESORI 74167, D. Comte, O. Gelly, J.C. Syre, ONERA - CERT, Toulouse, G. Durrieu, A. Plas, Université Paul Sabatier, Toulouse.

Authors'address : ONERA CERT / DERI
BP 4025
31055 TOULOUSE CEDEX FRANCE

If one makes a program obeying this semantic rule on data objects, then one has naturally expressed the data dependencies between the program statements, in a deterministic way. Given the fact that, once computed, an object value is unique and will not change any longer, then a new type of statement sequencing and execution is allowed :
- A statement is claimed "ready for execution" as soon as its operands have their values.
- It can be actually executed at any time later.

Once executed, the above statement produces values to output objects and will have to propagate that knowledge to other statements using $\emptyset$, or part of it, as operands. The fairly simple example listed below shows this new sequencing mechanism.

A=2	S1
C=A-B	S2
D=A+B	S3
E=A-1;	S4
F=(C*D)/(A*B)	S5
G=(C-D)/E	S6
H=B*(A+1)	S7
I=E*(A/2+H)	S8
B=10	S9

```
At first step of execution : S1//S9 (one or both
                                      of them)
At second step of execution : S2//S3//S4//S7
At third step of execution : S5//S6//S8
Execution sequence : (S1//S9),(S2//S3//S4//S7)
                     (S5//S6//S8)
Other possible sequence : (S1),(S9//S4),(S2//S3),
                          (S5//S6//S7) ; (S8)
```

Data dependencies can also be represented by the following data flow graph.

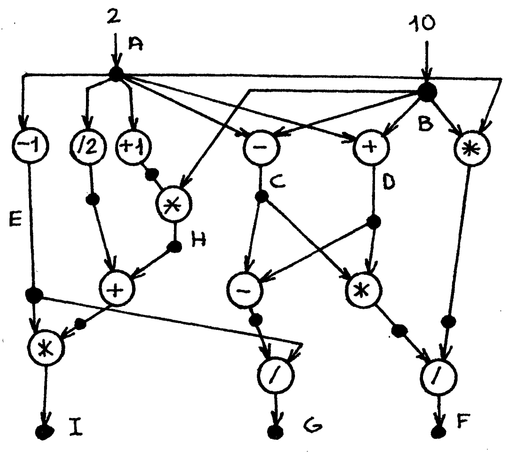


Fig 1 : Single assignment programming example.

All instructions, despite of their syntax, must be considered as assignment instructions. In order to enlighten this point, we now define first the general data and instruction formats. Then, the concept of Data Production Set is defined together with the different primitive operations acting on it.
After that we shall give, for each type of instructions,
- its syntax
- its semantic definition using the concept of Data Production Set (DPS)
- its execution using the primitive operations on the Data Production Sets (DPS).

General instruction and data formats

The general instruction format is composed of a result address, an operation code, two operand addresses and three control tag bits.

CODOP	RES	OP1	OP2		CiO	Ci1	Ci2

Fig 2 : General Instruction Format

The three control tag bits CiO Ci1 Ci2 denote the state of the instruction. Ci1 and Ci2 tell whether the two operands OP1 and OP2 are "known" or not. The third one tells about the environment control of the instruction (due to the possible nesting of the instruction within a DPS). This bit is set by the control instructions. An instruction will be executable, or "ready" when the three control tag bits match the 111 value.
The general data format is the following :

value	link1	link2		Cd

Fig 3 : General data format

An data object is composed of two kinds of fields : - a conventional value field
- several propagation fields
- a control tag bit.

Propagation fields are link1, and link2. This means that instructions at addresses link1 and link2 use the operand. A bit, in link address, denotes if it is a right or left operand. Additionnal links may be placed in the following words, if more than two instructions use the operand (compilation of real programs has shown that the average number of links is approximately two). The tag bit Cd denotes whether the data has been calculated or not. The lack of register addressing certainly implies a larger memory use of temporary data. However, the instruction execution does not depend on the processor's type or number, which will make the architecture completely modular and asynchronous. Only the operation code will be possibly used to drive a ready instruction on to a functional processor.

The concept of Data Production Set (DPS)

A DPS is defined as a couple of
- a set of instructions, I
- a set of data objects, O.

Any instruction will be completely defined by
- one or more DPSs
- how it will operate on its DPSs.
The different operations on DPSs are now given. They will be implemented in the Control Part of the processor structure and are considered as basic control primitives by the different instructions.

Basic control primitives

The four following primitives will act either on the I part or on the O part of a DPS, by activating or updating instructions tag bits, or by checking or updating data tag bits

P1 : SET TAG BITS (CiO, A, L).

This primitive sets to 1 the CiO tag bit from address A, of length L. (Activation of a DPS).

P2 : SET TAG BITS (Ci1, Ci2, A1, A2, L)

This primitive sets the control tag bits Ci1 and Ci2, with a boolean mask (built at compile time and stored at address A1), from address A2, of length L. This mechanism permits to clear the control tag bits when instructions are executed more than once (DPS clearing).

P3 : CHECK TAG BITS (Cd, A1, A2, A3)

This primitive checks the value of Cd from address A1 to address A2 and propagates, when all bits match '1' value, the event "end of checking" at address A3. This mechanism permits to know the end of a DPS activation (DPS termination).

P4 : MASK TAG BITS (Cd, A1, A2, L)

This primitive sets tag bits Cd, with a boolean mask (built at compile time and stored at address A1), from address A2, of length L. This mechanism permits to initialize the Cd bits of the object part of a DPS (starting at A2, of length L).
We now come to the different instructions of the machine language.

Computational instructions

Their syntax is exactly given by Fig. 2. They are semantically defined by a simple DPS. For example, C=A-B is implemented by the following instruction :

–	C address	A address	B address		1	0	0

The DPS created is composed of
- I : this instruction
- O : C

The execution of a computational instruction consists of :
1. reading OP1, OP2
2. performing f (OP1, OP2)
3. writing RES = f (OP1, OP2)
4. propagating the result by updating the tag bit Ci1 or Ci2 of instructions using RES, by means of link1, link2 ...

5. setting the tag bit Cd corresponding to RES.

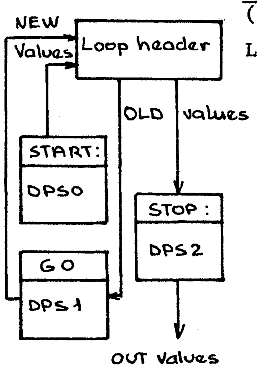
Control instructions

These instructions are close to the high level language.

ACT instruction : it operates on one DPS, and controls its execution. ACT makes use of :

P4 : DPS clearing (0 control part initialization)
P1 : DPS activation (I control part activation)
P3 : DPS checking (for DPS termination).

LOOP instruction : the general iteration process is expressed by the LOOP statement in the high level language, and by a set of machine instructions. We first give an idea of the semantics of the LOOP statement, and then we shall explicit its execution in terms of DPSs and primitives.



Example of LOOP statement
(computation of factorial)

```
LOOP L
 OUT:FACT; whitin the LOOP,
           FACT denotes NEW
           FACT
 LOCAL:I;
 (START):FACT=1;I=1;L=GO;
 (GO):I=OLD I+1;
        CASE(OLD I>N):STOP
        LOOP L;
            (ELSE): FACT=
        OLD FACT*OLD I;;
        END CASE;
 END LOOP L;
```

Fig 4 : LOOP statement and associated control structure

Each object X declared in $\{\text{OUT}\} \cup \{\text{LOCAL}\}$ section is split into 3 objects : OLD X will refer to the previous value of X, for the current iteration. NEW X, or simply X, will be the object computed by the current iteration. OUT X or X is the actual value of X as assigned by the LOOP statement. A LOOP Control event is first set to START, and is assigned at each iteration. The Loop Header is then activated and will activate itself the DPS corresponding to the value of the Loop event given by the previous DPS activation. When activated, a DPS produces the NEW objects values, then gives control back to the Loop Header. A special STOP value given to the Loop event will be interpreted by the Loop Header as the activation of a special implicit DPS STOP which will assign the OUT objects (actual objects computed by the Loop statement).
As far as its implementation is concerned, the Loop statement may be represented by the following algorithm :
1 - The START DPS is activated by an ACT instruction ; the NEW objects receive their values.
2 - A SWITCH DPS is then activated, pushing the NEW values into the OLD values.
3 - The LOOP instruction is executed. Depending upon the loop event value, it activates either a user defined DPS, or the STOP DPS. An activated user defined DPS will in turn activate the SWITCH DPS as in step 2, and the iteration process goes on.

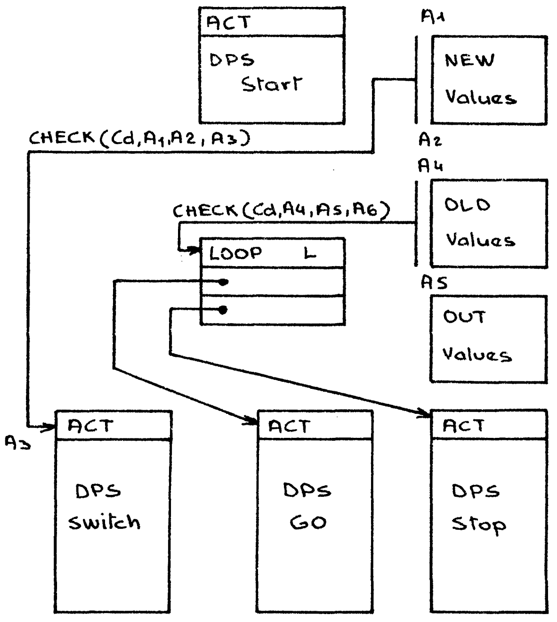


Fig 5 : LOOP internal mechanisms

CASE instruction : The CASE instruction, when all booleans are calculated, activates the DPS corresponding to the TRUE boolean and forces the production of objects which are not calculated.
The sequence of primitives is :
1. P1
2. P4

Exemple of CASE programming

```
CASE X
   (X=0):Y=2;
   (X=1):Y=3;
   (X>1):Y=4;Z=3;
   (ELSE):Z=1;
END CASE ;
```

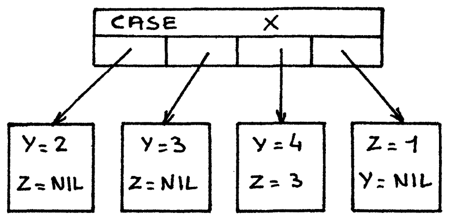


Fig 6 : CASE statement mechanism

EXPAND statement : The body of Expand forms a DPS. As each iteration is independent from the others, we can execute several such DPSs concurrently. The number of copies is static, but can be fixed by the programmer, according to the level of parallelism he wishes. An ACT instruction is associated with each copy. The EXPAND mechanism splits into two instructions : STEXP (START EXPAND) and EXP (EXPAND). There is one instruction STEXP, and n EXP instructions, according to the number of copies. The STEXP is an initialization instruction, i.e., it initializes the index of each copy, activates the corresponding EXP instruction and checks for the end of the n copies.
The EXP instruction controls its own copy : it first clears the control tag bits Ci1 Ci2 by means of (P2), increments and tests the current index, and activates the ACT instruction (if the index is less than the upper bound) associated to the copy.

Example of EXPAND programming

```
EXPAND I=A STEP B TO N:
       TAB (I)=X+I;
END EXPAND;
```

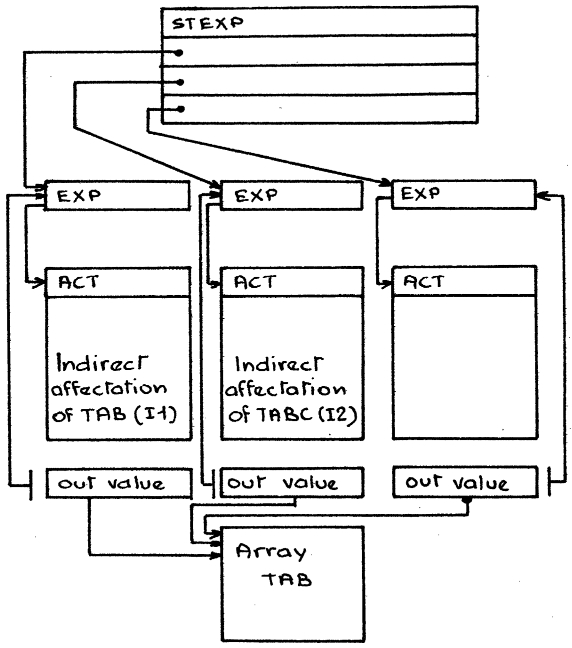

STEXP
EXP
EXP
EXP
ACT
ACT
ACT
Indirect affectation of TAB (I1)
Indirect affectation of TABC (I2)
out value
out value
out value
Array TAB

Fig 7 : EXPAND mechanism

CALL Statement : The body of the procedure is a DPS with inputs and outputs which are formal parameters. Several copies of the procedure are generated by the compiler on the request of the programmer. A header, associated with the procedure, manages the calls.

When actual input parameters are calculated, the CALL instruction becomes executable and looks for an idle copy by means of the header. If there is one, formal parameters are assigned with actual parameters and the copy is activated by the ACT instruction corresponding ; if there is no idle copy, the CALL instruction is put into a waiting queue.
A RETURN Instruction, when all outputs of the copy are calculated, is activated. It tests the content of the queue, and if there is a waiting CALL, releases it.
This mechanism can be extended to non standard objects. The synchronization algorithm, given by the programmer, can be described in the header of the object and executed at every access to the object.

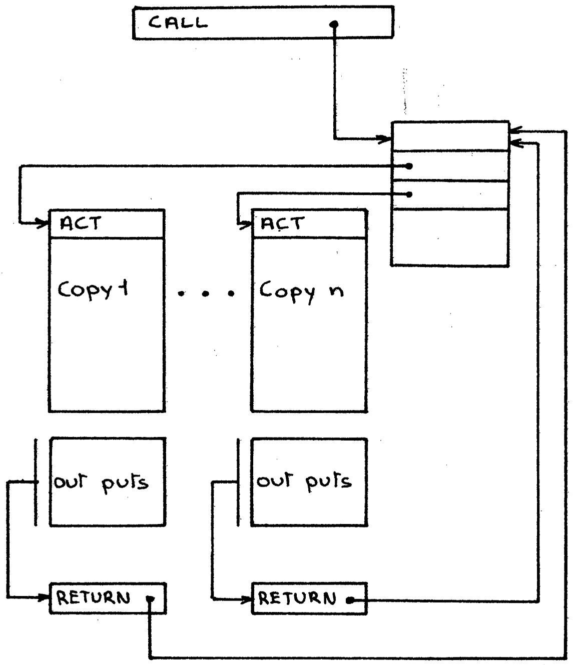


Fig 8 : CALL and RETURN Instructions

The general architecture

The processor now described takes place in a greater architecture and is only the execution part of the system. We give here a brief account of the general architecture.
(See Fig. 7).

The peripheral processors deal with peripheral devices. The secondary storage holds all of the programs users. They are managed by the Job Supervisor which knows which are the programs actually running together on the machine.
Programs are divided into a set of tasks by the compiler. These tasks are loaded into an execution processor on the request of the Scheduler contained in the Job Supervisor.

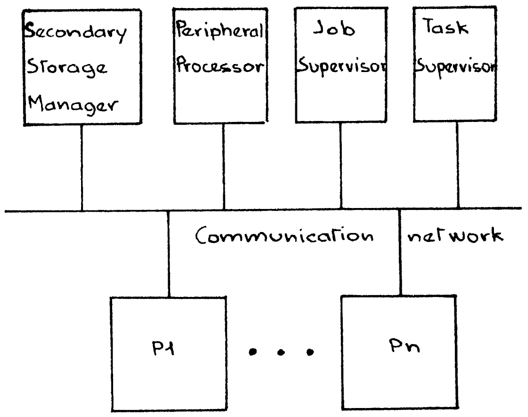


Fig 9 : General architecture

The execution processors are independent and are managed by the Task Supervisor which knows the state of each processor and informs the Job Supervisor which tasks must be swapped. This architecture allows three levels of parallelism :

- between the different programs (concurrent programming)
- between the different tasks (inter task parallelism)
- between the instructions within a task (elementary actions).

Now we shall insist on the last kind of parallelism provided naturally by single assignment.

The processor structure

The processor is composed of three units :

- the local memory
- the execution unit
- the control unit.

The local memory subsystem

The memory contains one or several tasks at a given time. Each task is composed of two parts : instructions and data, and can be considered as a DPS. A task is loaded when its inputs are known. The task will be said terminated when all its outputs are known (calculated or no longer calculated). Since the number of memory accesses is of great importance this memory has been divided into several interleaved banks managed by a Memory Control Unit (MCU).

The MCU controls all memory accesses coming from the Execution Unit and the Control Unit. Memory conflicts are solved using a priority mechanism. The MCU dispatches the information coming from memory towards the right unit.

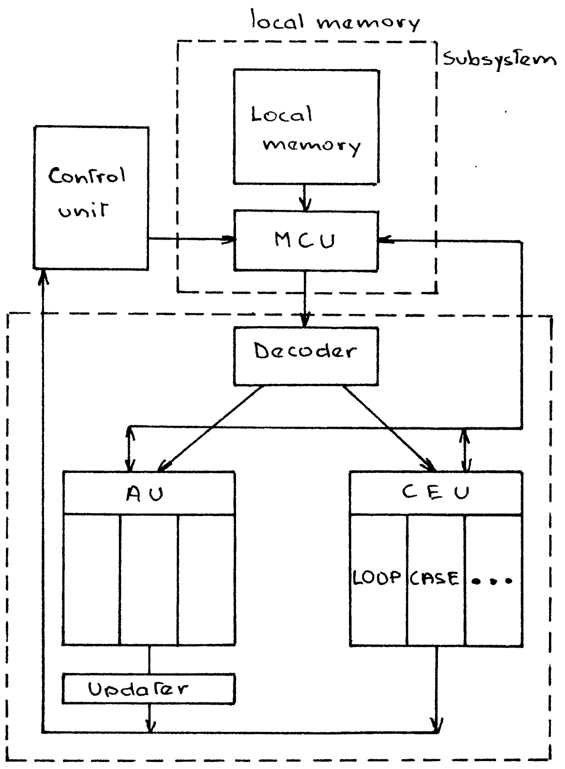


Fig 10 : The processor structure

The control unit (CU)

This unit is the truly original part of the processor. It implements the typical sequencing mechanisms of a data driven execution. It holds the control which indicates the state of the machine (instruction and data). Two memories hold the control bits Cd and (Ci0, Ci1, Ci2).
The Instruction Control Memory (ICM) contains the instruction control tag bits (Ci0 Ci1 Ci2).
The Data Control Memory (DCM) contains the data control tag bits (Cd).

The Instruction Control Memory. The ICM is composed of n three bit wide words, where n is the length of local memory. A word at address x is associated with a word in local memory, becomes executable when the three bits match the '111' configuration. Two devices act on this memory.

- The ICM Updater Processor (ICMUP) modifies the contents of the ICM by executing the functions sent by the Execution Unit. The ICMUP can set any of the three bits Ci0 Ci1 Ci2 without modifying the others.

This needs two accesses to the ICM : the information is read from the memory, modified in the ICMUP and finally written back into the memory. The ICMUP can perform several logic operations (OR, AND) between the informations received from the bus and the information read from the ICM.

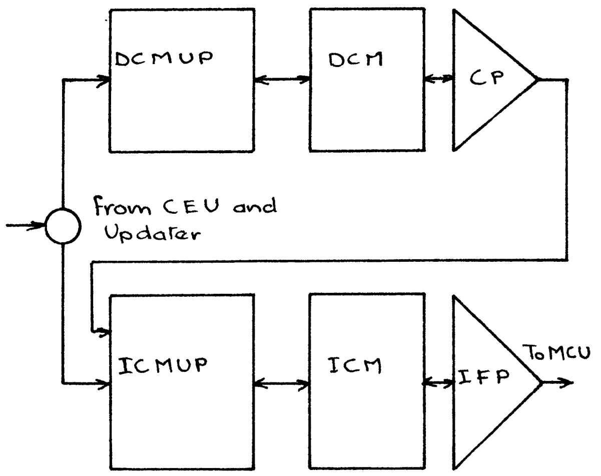


Fig 11 : The control unit.

- The Instruction Fetch Processor (IFP) checks for ready instructions. This processor has an associative access to the ICM (ICM is a content adressable memory). When finding the '111' configuration it sends the corresponding address to the MCU and asks the ICMUP to write the '011' value in the word matching '111'. (Not to find again the ready instruction). The MCU reads the ready instruction from the local memory and sends it to the Execution Unit. IFP has a lower priority than ICMUP, as far as ICM accesses are concerned.

The Data Control Memory. The DCM is a one-bit wide memory of length n. Cd set to 1 at address x means that the datum, at address x in the local memory, is calculated or will never be calculated. Like the ICM, the ICM is associated with two devices acting on it.

- The DCM Updater Processor (DCMUP). Its role is the same that the ICMUP, i.e. it executes, on DCM, the updating functions sent by the Execution Unit.

- The Check Processor (CP). This device executes the Primitive (P3) : CHECK TAG BITS (Cd, A1, A2, A3). It checks the DCM from address A1 to address A2. If all Cd bits are '1', it asks the ICMUP to write the '111' configuration at address A3, i.e. its make the associated instruction executable. If a Cd bit is not '1', the DCMUP pushes the Primitive into a waiting queue and releases another primitive of the queue for execution.

The Execution Unit.

The Execution Unit is multi-pipeline unit. Due to the data driven control, ready instructions are independent, so the computation power may be split into several asynchronous units, each of them being able to execute ready instructions in parallel. This unit is composed of three parts :
- A decoder
- An Arithmetic Execution Unit (AU)
- A Control Execution Unit (CEU)

The decoder dispatches ready instructions towards the right Unit. It works on a buffer which contains instructions. This buffer is supplied with instructions by the MCU on the request of the IFP. The IFP has a variable priority to access local memory, this priority is managed by the decoder. When the content of the ready instruction buffer is greater than an upper bound the priority of IFP is decreased ; if the buffer is full, IFP is stopped. This decoder analyzes the first bits of the operation code and sends the instruction to AU or CEU.

The Arithmetic execution unit (AU) : This unit comprises several subunits :
- floating point execution unit.
- fixed point execution unit.
- vector execution unit.

To improve parallelism, there may be several identical subunits. Pipelining floating point and vector units is also a good means to improve concurrency because pipeline mode of execution is well adapted in a data-driven processor. Since no dependency exists between ready instructions, these instructions may enter the pipeline in any order and they will certainly deliver a result. Hence all precedence conflicts are suppressed, and so are the gaps caused by conditional jumping for the simple reason there is no branching instruction.
An updater is associated with the Arithmetic Unit. It executes the steps 4 and 5 of the execution : when the result is calculated, the updater receives link1 and link2 from memory, and acts on the control unit to update the corresponding tag bits.

The control execution unit (CEU) : The CEU is divided into independent functional subunits. Each subunit performs a control instruction. We can find the LOOP, CASE, EXPAND, CALL, ACT subunits. As in the AU, a subunit may be duplicated to improve parallelism. It may be the case of EXP subunit, because there are always several EXP instructions concurrently executable in the task.

Performance - Simulation

Parallelism achievable in the processor depends on the power of the execution unit (number of operators), but it also depends on the level of parallelism we can find in programs. Simulation has shown that most of programs contain parallelism (see simulation results). The programmer may adjust the level of parallelism by changing the number of copies (EXPAND instructions) generated at compile time.

A simulator of the processor has been designed. It makes use of parallel processes facilities offered by a software system built formerly. Each elementary unit described in the paper is an asynchronous process. This simulator works on real programs generated by a compiler, from the high level single assignment language. The simulator shows us, by comparison with real sequential machines, that performances are good if the level of parallelism is great enough,(that is the case of many programs).

Due to constraints imposed by our computer facility, the simulator allows no more than 12 A.U processors and 8 Control processors.
The standard implicit values for the different parameters of the simulated LAU system are the following :

- Control Unit subsystem : Instructions (CO C1 C2): One memory bank, 50 ns cycle time, Data (Cd) : one memory bank, 50 ns cycle time. One checker processor.
- Memory subsystem : 4 banks, 500 ns cycle time, 100 ns manager cycle time.
- Processing subsystem : 12 AU processors, 2 EXP ,CASE,LOOP,ACT processors. Typical add time : 100 ns.

This LAU system corresponds to "STD00" simulations. 34 other configurations have been taken to measure the influence of the LAU architecture on program execution, some of them being listed on Fig 12. About 50 programs have been executed for every configuration. Only partial results can be shown here, however they may give an idea of the overall performance of the LAU system.

EXEC NAME	CONTROL UNIT SUBSYSTEM					MEMORY SUBSYSTEM			PROCESSING SUBSYSTEM				
	# BANKS INST.	CYCLE TIME INSTRUCTIONS	# BANKS DATA	CYCLE TIME DATA	CHECKING PROCESSOR	# BANKS	CYCLE TIME ns	MANAGER CYCLETIME	# A/L	# EXP	# ACT	# CASE	# LOOP
STD 00	1	50	1	50	1	4	500	100	12	2	2	2	2
STD 1									12	4	4	2	2
2			as	above					12	6	2	2	2
3									10	4	4	2	2
4									10	6	2	2	2
5									10	8	2	2	2
PCTL 2	1	50	1	50	2	4	500	100	12	2	2	2	2
PCTL 3		05	abo	ve	3			as	abo	ve			
4					4								
5					5								
8					8								
MEM 1	1	50	1	50	1	4	500	100	12	2	2	2	2
3						1	100	100					
4						2	250	100		as	above		
5		as	above			2	600	100					
6						2	200	100					
10						8	800	100					
12						1	160	160					
13						2	320	160					
14						4	650	160					
UCTL 1	1	30	1	30	1	4	500	100	12	2	2	2	2
UCTL 2	1	60	1	60	1			as ab	ove				
UCTL 9	4	120	4	120	1								

Fig 12 : Part of configurations list for LAU system simulation.

Evaluation of the EXPAND statement Fig 13 shows execution times for two programs. SMEX is an EXPAND statements whose body consists of one array element instruction, BGEX is an EXPAND statement whose body contains 22 instructions. Fig 14 - 15 give the parallelism observed in the machine.

SMEX exec. time (μs)							
DEGREE OF EXPANSION	1	2	4	6	8	12	16
OBJECT CODE (HEXA)		19	29	3B	4B	6D	8F
STD 00	1100	639	364	299	306	318	336
STD 1	"	"	348	286	259	233	251
STD 2	"	"	"	278	246	$\boxed{229}$	240
STD 3	"	"	"	286		233	251
STD 4	"	"	"	278		229	240
STD 5	"	"	"	"	248	241	236

BGEX exec. time (μs)							
DEGREE OF EXPANSION	1	2	4	6	8	12	16
STD 00		2670	1809	1637	1556	1554	1559
STD 1		"	1780	1621	1548	1530	
STD 2		"	1794	1611	1557	1516	
STD 3		"	"	1648	1535		1522
STD 4		"	"	1628	1531	$\boxed{1513}$	1541
STD 5		"	"	"		1546	1543

in μs	MEM 10	~ IBM 370/148
SMEX(12)	394.20	1530
BGEX(12)	1709.90	6000

Fig 13 : SMEX and BGEX executions.

These results show that :

- In SMEX, overhead is very important, and the optimal expansion degree is nearly 6. A higher degree makes object code too big and corresponding execution not faster than expected. Parallelism is interesting for SMEX 8 (3), SMEX 12 (6), but not for SMEX 16 (8).
- In BGEX, there is much more "good" parallelism. The larger the object code, the more parallel execution is, but some factors must be taken into account : First, the length of object code becomes important for degrees of expansion larger than 8 or 12. Secondly, the processing subsystem is saturated beyond 8 parallel expansions, and increasing the expansion degree is no more useful for the configurations chosen.

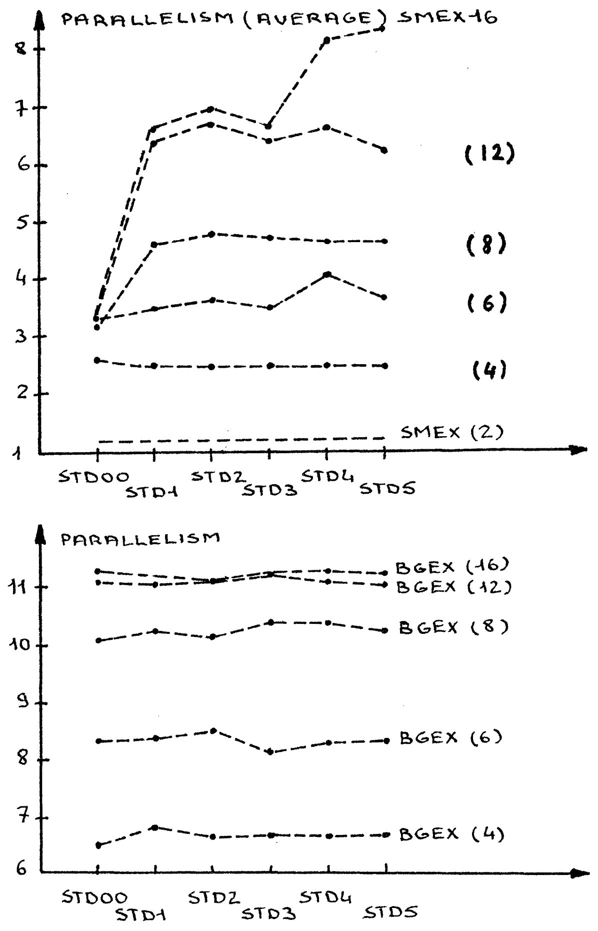


Fig 14 : Average parallelism for SMEX and BGEX (depending on expansion degree, and configuration.

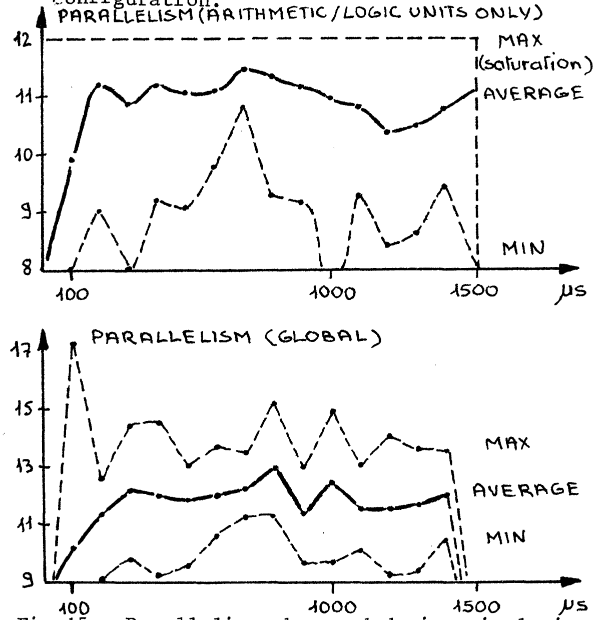


Fig 15 : Parallelism observed during simulation STD2 of program BGEX (12).

- Program CARLXX simulation results. This program computes $\int_0^1 f(x)\,dx$ by the Monte Carlo method. Here $f(x) = x^2$.
For convenience, the random numbers ALEAX and ALEAY are put into two arrays at run time.
The source program is as follows

```
%/C/CLEAU/LS,LO,CM,XD,EX(4)
PROGRAM CARLX040G
DECLARE:
  ALEAX,ALEAY:ARRAY(0:15) OF INPUT;
  CARLO : ARRAY(0:15) OF INTEGER
  NX,INTEGRALE : INTEGER;
  XY,YC,I1 : INTEGER;
DO:
  NX=15
  EXPAND I1 = 0 TO NX :
      LOCAL : Y,YC,X
      X = ALEAX (I1);
      Y = ALEAY (I1);
      YC = X*X;
      CARLO (I1) = (YC>Y);
  END EXPAND;
  INTEGRALE = VSUM(CARLO)/(NX+I);
  OUTPUT 1 : INTEGRALE;
END PROGRAM.
```

By changing the expansion degree as declared in the compile command, 3 programs have been generated CARLX040G, CARLX080G, CARLX160G.
Fig 16 lists the static data for these programs.

	CARLX04	CARLX08	CARLX16
SOURCE CODE	27	27	27
OBJECT CODE	158	257	356
TEMPORARY OBJECTS	32	73	114

Fig 16 : CARLXOG : Static data

As far as dynamic data are concerned, the two curves below can summarize the behaviour of these programs, depending on the simulation parameters.

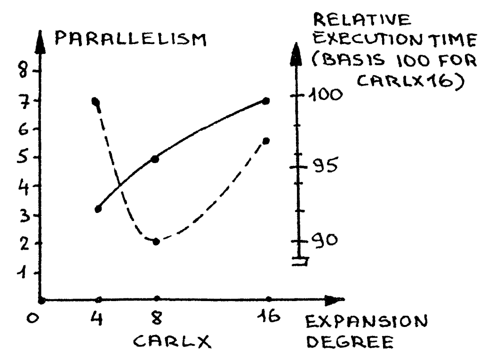


Fig 17 : Parallelism and relative execution times for STDOO simulation.
Fig 17 shows that, despite a high degree of parallelism CARLX160G suffers from some conflicts in ressource sharing, Fig 18 confirms this remark when we are interested in the different memory subsystem configurations.

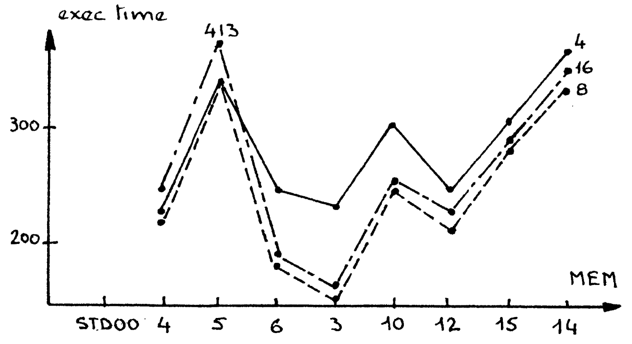


Fig 18 : CARLX : Exec times versus Memory subsystem configuration.

Other simulation results. Since our machine is not dedicated for a given application, we have experimented various programs such as sorting algorithms, pattern matching problems or mathematical formulas. In Fig 19, PLAGOO denotes the Lagrange polynomial computation, TRIP05 denotes a parallel sorting algorithm. PMAT30 is a 3x3 matrix multiply POLYOO is a polynom product. PAYE02 is a program computing salaries and incomes in our company.
Fig 19 shows the results for various executions and the corresponding times on a CII IRIS 80 (similar to an IBM 370/145).
Notice that source programs have been written without any optimization techniques (which will be used later).

PROGRAM	EXEC NAME				IRIS
NAME	STDOO	MEM3	UCTL1	PCTL4	80
POLYOO(4)	1.402	0.905	1.385	1.349	3.6ms
PMAT30 (3)	0.546	0.392	0.541	0.514	1.6ms
PLAGOO (2)	1.595	1.102	1.531	1.532	12.6ms
TRIPO1 (4)	3.052	2.204	3.079	3.021	4.8ms
PAYE 02	1.396	0.913	1.319	1.366	2.4ms
CAPLX 08	0.226	0.158	0.216	0.214	0.360

Fig 19 : Other simulation results and comparison to sequential executions on CII IRIS 80.

Conclusion. Though a great deal of work is still to be done, the LAU system simulation results allow us to say that single assignment may be a good way to express and exploit parallelism in programs. Further work should confirm that. By now, results have given us some hope for the future of our project.

Conclusion

This paper has presented the outlines of the LAU parallel system. Based on the software concept of single assignment, this system is composed of :

- A high level language, close to Pascal, which allows the user to express the concurrency in the program statements in a natural way. Parallelism is achieved by the data dependencies only. All statements, including decision or iteration statements are considered as assignment statements, due to the concept of Data Production Set.
This high level language has been tested on a large number of problems.
- A machine language, based on a three-address instruction format. This language implies entirely new cadencing mechanisms based on data directed execution of instructions, and can be executed on a completely modular asynchronous multiprocessor structure. A compiler translates programs written in the high level language into executable code, and includes many debugging tools.
- A multiprocessor architecture, whose Control subsystem is unique and implements the control primitives on data and instructions tag fields. Its main characteristics are modularity, pipelining and asynchronism allowed by the data driven mode of execution. The corresponding simulator has been fully parametrized and some partial results have been given here.

The next step in our study will be to optimize and simplify the LAU multiprocessor, and build a prototype which could be specialized for some classes of applications.

References

[1] J.C. Syre, et al, Parallelism, Control and Synchronization expression in a single assignment language, 4th ACM Computer Science Conference Anaheim, Feb. 76.

[2] O. Gelly et al, LAU Software System : A high level data driven language for parallel programming these proceedings.

[3] D. Comte, et al, The LAU parallel system : software definition and implementation through a multiprocessor architecture. EUROMICRO 2nd symposium on microprocessing and microprogramming. October 12 - 14 Venise 1976.

[4] Tesler L.G., Enea H.J. A language design for concurrent processes. Proc. AFIPS, SJCC Vol 32, 1968 pp 403 - 408.

[5] Chamberlin, D.D. Parallel Implementation of a single assignment language Ph. D. thesis January 71 - Technical report n° 13.

[6] Klinkhamer, J.F. A definitional language, Philips Research Laboratory, Internal report Eindhoven (Hollande), 71.

[7] Irani K.B., a proposal for a programming language for parallel processing environments, Internal Report Jul 72 - Jan 75, RADC, Febr. 75.

[8] Urschler, G., The transformation of flow diagrams into maximally parallel form. Sagamore conference on parallel processing, 1973.

[9] Dennis, J.B. 1st version of a data flow procedure language, computation structures group MEMO 93, MIT, Nov. 73.

[10] Dennis, J.B. Packet Communication Architecture, Sagamore computer conference on Parallel Processing, 1975 p. 224 - 230.

[11] Misunas, D.P., Performance of an elementary data flow processor. Computation structures group MEMO 115, Project MAC, MIT February 1975.

[12] Ramamoorthy C.V., and Li, H.F., Pipeline processor architecture. A survey, Sagamore computer conference on parallel processing 1975. p. 40 - 63.

[13] Rumbaugh J.E., A parallel asynchronous computer architecture for data flow programs, Ph. D. thesis. MIT Project MAC. May 1975.

[14] Comte D., Durrieu G., Gelly O., Plas A., Syre J.C., Techniques et Exploitation de l'Assignation Unique. Vol 5 - 8. Contract SESORI 74 167.

[15] Anderson, D.W., Sparcio F.J., Tomasulo R.M. IBM System 360/91 : Machine philosophy and instruction handling. IBM Journal, Vol 11, N° 1, 1976 p 8. 24.

UPPER BOUNDS ON THE PERFORMANCE OF SOME PROCESSOR-MEMORY INTERCONNECTIONS

R.C. Pearce and J.C. Majithia
Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, Canada

Summary

The maximum multiprocessor performance achievable is evaluated for four commonly proposed switching methods: cross-point, time-shared bus, pipelined loop, and binary switch. The processor behaviour model is based on a "unit instruction," consisting of a single memory access followed by a data processing interval. The upper bound on performance is evaluated under the assumption that the number of processors and memory banks are equal, the memory request pattern produces no conflicts, and the arbitration time for conflict resolution is negligible.

The performance of a cross-point switch [1] is affected by two switching delays, one for transmission of the address to the memory and one for return of the data to the processor. Hence, the multiprocessor cross-point throughput is given by

$$T_c = \frac{n}{t_p + t_m + 2t_s}$$

where n is the number of processors and memories, t_p is the processing time per unit instruction, t_m is the memory access time and t_s is the switching delay per stage. Consider a time-shared bus multiprocessor [2] which makes use of two busses, an address bus, for transmission of addresses to the memories, and a data bus for return of data to the processors. The time-shared busses will become saturated when the number of processors N is such that $Nt_s = t_p + t_m + 2t_s$. Hence, the time-shared throughput is given by

$$T_t = \frac{n}{t_p + t_m + 2t_s} \qquad \text{for } n \le N$$

$$= \frac{1}{t_s} \qquad \text{for } n > N$$

A multiprocessor could also be operated with a pipelined loop as the main data path. Each "slot" on the loop would either be a memory request or a data return for the processors. The maximum throughput for the pipelined loop is

$$T_\ell = \frac{n}{t_p + t_m + 2nt_s}$$

where 2n is the number of loop nodes, one for each processor and memory. The binary switch [3] is arranged in $\log_2 n$ stages where n is the number of processors and memories. Each of two incoming lines to a binary switch module can be connected to either of two output lines. Routing is accomplished by using one bit of the memory bank address for selection at each stage. The maximum throughput attainable by the binary switch is

$$T_b = \frac{n}{t_p + t_m + (2\log_2 n)t_s}$$

where $\log_2 n$ is the number of switch stages.

The normalized performances are shown on the graph below for example values t_p = 500nsec, t_m = 500nsec, and t_s = 100nsec. Simulations were also done with the conflict-free assumption removed and these showed the same relative behaviour between the four switching methods but at reduced performance. The crosspoint switch shows a linear increase in performance for a cost increase of n^2, while the binary switch shows a $n/\log_2 n$ increase in performance for a cost of $n\log_2 n$. The time-shared bus shows a linear performance until the bus bandwidth is reached. The pipelined loop shows poor performance in comparison to the other techniques. The results indicate that the most cost effective solution for a small number of processors would be a time-shared bus while for a large number of processors the binary switch yields the highest throughput for the least cost. Further studies are being made into improving the performance characteristics of the binary switch for use in a multiprocessor computer architecture.

References

[1] W.A. Wulf and C.G. Bell, "C.mmp - A Multi-Miniprocessor", AFIPS FJCC Proceedings, 1972, Vol. 41, Part II.

[2] P. Danielsson and B. Gudmunsson, "Time-shared Memory-Processor Interface", Proceedings of the 1975 Sagamore Conference on Parallel Processing.

[3] I.A. Davidson and J.A. Field, "Design Criteria for a Switch for a Multiprocessor Computing System", Proceedings of the 1975 Sagamore Conference on Parallel Processing.

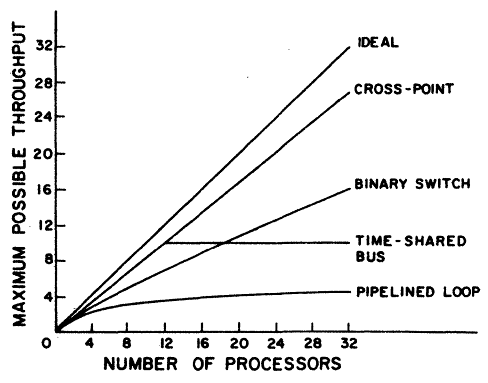


Fig. 6 PERFORMANCE (UPPER BOUND) OF VARIOUS MULTIPROCESSOR SWITCHES

OPERATING SYSTEM MODELLED AS A CONGLOMERATE OF INTERDEPENDENT ACTIVITIES

T. Feng
Department of Electrical & Computer Engineering
Wayne State University
Detroit, Michigan 48202

C. P. Hsieh
Undersea Electronics Programs Department
General Electric Co.
Syracuse, New York 13201

Abstract -- A computing system is viewed upon as a collection of different resource types to serve different users with different demands, while its operating system assumes a managerial role. To best utilize the available resources to achieve a desirable level of production, i.e., computation, an optimal planning (programming) is needed. Optimality can be judged if a performance index can be established and this index can be quite general. The question of system resource allocation is then formulated as a linear programming problem with constraints on resources, and optimization is over a linear objective function. Program loading (memory allocation) is static while program execution (scheduling, or dispatching) is dynamic in a multiprogrammed environment. The scheduling problem is studied through the viewpoint of memory utilization with a warehousing model.

The Programming of Activities

The notion of programming is a general one. On the user level, an individual program can be considered as the organization of activities, which, when successfully carried out, would achieve the objective of a computation. The trend of using high level language removes a user from the details of resource management. In fact, he is oblivious to them. On the system level, the main concern would be the proper coordination of individual user's activities under the limitation of system resources. It is the programming of the latter kind that we will be dealing with in the ensuing discussions.

Basic Assumptions

An operating system may be considered as comprised of various observed activities. We may also assume that there exist some refinements as representative building blocks of different types that might be recombined in varying amounts to form yet more complex but possible activities. The whole set of possible activities will be referred to as a technology, i.e., the technology of operating systems. Additional assumptions that are closely related to those postulated by Dantzig in the study of econometrics [1], may be made as follows:

(1) There exists a set of all possible activities.

(2) There exists a finite set of basic activities, x_i, such that any possible state of an activity can be represented as

$$\sum_i a_i x_i \qquad i = 1, 2, \ldots, n$$

where a_i is the level of basic activity x_i.

(3) There exists a linear objective function,

$$\sum_i c_i x_i \qquad i = 1, 2, \ldots, n$$

where c_i is a constant associated with x_i, depending upon a specific formulation of the objective function.

(4) An activity in a possible state consumes a certain amount of resources of a certain kind, possibly limited by a constant b. That is,

$$\sum_i a_i x_i \leq b \qquad i = 1, 2, \ldots, n$$

The Allocation Activity

The immediate task is to identify a finite set of basic activities. Since a resource allocator deals exclusively with user programs, it appears natural to choose the set of user programs on the system job queue as the set of basic activities. More precisely, since a user's program may consist of more than one stage (job step), it is that particular job step that is up for allocation consideration that becomes one component of this basis. If we further perceive that each possible state may consume different kinds of resources, and that if we adjoin these possible states together, we have

$$Ax \leq b \tag{1}$$

where A is a rectangular matrix, x and b are column vectors. In reality, the number of elements in column vector b is equal to the total different resource requirements and other constraints by the set of basic activities, the column vector x. Thus, the allocation activity becomes the finding of a solution to Eq. (1). Since the feasible solutions to Eq. (1) are many, we may naturally want to find the most desirable one according to some criterion. We have thus come to the notion of goal oriented allocation. That is, if we further establish a linear objective function and set our goal to be the vector x that satisfies Eq. (1) and that also maximizes the objective function. This is stated formally as follows:

$$\begin{aligned} &\text{Maximize:} \quad \sum_i c_i x_i \qquad i = 1, 2, \ldots, n \\ &\text{Subject to:} \quad Ax \leq b \end{aligned} \tag{2}$$

Program Loading - Static Planning

Although a program may require many different types of resources before the execution can be commenced, none will be more critical than memory space. Thus, we choose to look at the system allocation activity as primarily, at least for the moment, an activity which distributes primary commodities (memory spaces) among the basic activities (individual user programs) to achieve productions (computations). The purpose is then to devise a way (a plan) to allocate those available memory spaces such that the computer system may execute programs according to some policy. In a simplified viewpoint, we equate the allocation of memory space to a program to that of initiating that particular program from system job queue to system ready queue. Using a common, long-established terminology, we would say that this is "loading" a program into the memory. If we consider that the loading action happens at discrete points in time and that at each occurrence of this action, memory will be filled to the extent possible, according to some goal, then this action is static in nature. That is, the goal is either satisfied or not, at the moment of loading, and not over a period of time. If we state our criterion in the form of a linear objective function, the simplex method provides the answer.

Problem Formulation

Let x_i be a fraction of an individual program i, i.e.,

$$0 \leq x_i \leq 1 \qquad (3)$$

and $(x_1, x_2, \ldots, x_n)$ the collection of such fractional programs. We look upon x_i as a basic activity, and if associated with x_i there is a number λ_i, the "level" of the activity, then the total activity, i.e., the allocation activity, is constrained by the total resources, M. It is

$$\lambda_1 x_1 + \lambda_2 x_2 + \ldots + \lambda_n x_n \leq M. \qquad (4)$$

If M is the total memory space available, then λ_i is the size (maximum memory units required per basic activity) of program i. We further state that the goal of our allocation activity is to plan our use of the memory spaces such that a certain linear function, namely, the objective function, is maximized. This objective function has the general form of

$$c_1 x_1 + c_2 x_2 + \ldots + c_n x_n, \qquad (5)$$

where c_i's are constants.

The canonical form of this maximization problem is the following:

$$\begin{aligned} &\text{Maximize:} && cx \\ &\text{Subject to:} && Ax \leq b \qquad (6) \\ & && 0 \leq x_i \leq 1, \end{aligned}$$

where

$$A = \begin{bmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_n \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ 0 & 0 & \cdots & 1 \end{bmatrix}, \quad b = \begin{bmatrix} M \\ 1 \\ \cdot \\ \cdot \\ \cdot \\ 1 \\ 1 \end{bmatrix}, \qquad (7)$$

c = a constant row vector.

We can devise a specific optimal allocation plan only if the objective function, i.e., the row vector c, is explicitly defined.

Example

Let us consider a set of programs (x_1, x_2, x_3) with memory requirements 40, 30, and 20, respectively, and a total memory equals to 60 units, i.e.,

Program	Size	Total Memory
x_1	40	M = 60
x_2	30	
x_3	20	

Note that the program size and the total memory are of the same units, such as words, blocks, or pages. Following Eq. (4), we can write:

$$40\, x_1 + 30\, x_2 + 20\, x_3 \leq 60 \qquad (8)$$

In addition to the resource constraint, we also have the condition stated in Eq. (3). Combining all the constraints, we may write down a set of simultaneous inequalities as follows:

$$\begin{aligned} 40\, x_1 + 30\, x_2 + 20\, x_3 &\leq 60 \\ x_1 \qquad\qquad &\leq 1 \\ x_2 \qquad &\leq 1 \qquad (9) \\ x_3 &\leq 1 \end{aligned}$$

Clearly, the structural matrix A and the constraint vector b assume the following values:

$$A = \begin{bmatrix} 40 & 30 & 20 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 60 \\ 1 \\ 1 \\ 1 \end{bmatrix}. \qquad (10)$$

What remains to be specified is our objective function.

A. Case I

If our objective for the memory allocation is to pack as many programs as possible, i.e., maximum degree of multi-programming, then we may write the objective function as:

$$\text{Maximize:} \quad x_1 + x_2 + x_3 \qquad (11)$$

The row vector c thus becomes:

$$c = [1, 1, 1] \qquad (12)$$

If x_i's are treated as continuous variables, then the solution vector $x^T = [\frac{1}{4}, 1, 1]$ satisfies the constraints of Eq. (9) and the requirement of Eq. (11). However, if we consider the programs x_i's to be indivisible, then we must look for integer solutions. In this case, we have two feasible solutions:

$$x^T = [0, 1, 1] \quad \text{or} \quad x^T = [1, 0, 1] \qquad (13)$$

B. Case II

Suppose, associated with each individual program, there is a "value". Specifically:

Program	Size	Value
x_1	40	70
x_2	30	50
x_3	20	30

Each value shown here is a quantification of the relative importance of each program. Then, Case I can be considered as a special case in that all programs are of equal importance. Let us state that the goal is to find a subset of programs to load into the memory such that the total values are at a maximum. The objective function becomes:

$$\text{Maximize: } 70\, x_1 + 50\, x_2 + 30\, x_3 \qquad (14)$$

Once again, if we restrict ourselves to integer solutions, $x^T[1, 0, 1]$ satisfies Eq. (9) and the requirements of Eq. (14).

Interpretations and Related Questions

In Case I, there are two possible solutions. Each can be considered as an optimal solution since each achieves arrived at these two solutions by different pivoting sequences. However, it is highly impractical for the operating system to set up and solve maximizing problems every time the system has to carry out the allocation activity.In other words, arriving at an optimum policy, i.e., finding the solution which solves the programming problem,is not necessarily the same as implementing it.

From practical considerations, solving the linear programming problem is a problem of sequentially loading the programs in accordance with the given constraints.

Branch-and-bound method [2] in solving the integer programming problem can, of course, handle this situation. This method essentially involves implicit enumeration on all the feasible solutions and chooses the solution that optimizes the objective function. When the number of variables involved is large, it becomes cumbersome. To avoid such cumbersome procedure we introduce the following heuristic approach which provides an alternate solution to the problem. It will yield an optimal solution probably most of the time, but not all the time, yet the procedure is much simplified.

Heuristic Approach

Let us assume that we have a pool of programs $(x_1,\ldots,x_n)$ which are to be considered for memory space allocation. Associated with each x_i, there is a value of c_i, and the value-to-size ratio can be formed. Let us further suppose that this community of programs are put into a sorted list according to the magnitude of each value-to-size ratio in descending order. This sorted list has n items with the top and the bottom each corresponding to the largest and the smallest ratio, respectively. The relative positions of, or the index to, this sorted list signifies the magnitude of each value-to-size ratio relative to each other. For $n \leq 2$, the sequencing problem is obvious, For $n \geq 3$, the algorithm is shown in the form of a flow chart in Figure 1.

i) At least one program can fit in.

ii) We are working with a sorted list, i.e.:

$$\left(\frac{c_k}{\lambda_k}\right)_i > \left(\frac{c_p}{\lambda_p}\right)_j \qquad \text{if } i > j$$

where i, j are indices (positions) on this list.

iii) In the flow chart, "load j" means to load the program that is in the jth position on the list.

Clearly, the algorithm as proposed is a suboptimal one.

The Value Concept

In our previous discussions, we have used the term "value" freely,without actually elaborating on it. Also, we have seen that under similar circumstances, the formulation of different objective functions could lead to different allocation plans. We have propounded the notion of goal oriented allocation. What is this goal? It is clear from the context that we have chosen our goal to be the maximization of a given set of possible values. The programs thus selected (allocated) would be of the utmost valuation to the system if actually processed. Therefore, the whole question of system performance is tied to the resource allocation problem through the determination of a general objective function. Furthermore, this general objective function can be formulated by defining a generalized value for each individual allocation unit, such as a job, or job step.

Definition: For each program unit x_i there is an associated generalized value c_i, such that

$$c_i = G\,(a_{1i}, a_{2i}, \ldots, a_{ki})$$

for $i = 1, 2, \ldots, n$, where a_{ji}, $j = 1, 2, \ldots, k$ are the individual attributes of program i and G is any well defined function or a composite of functions.

As is defined, the function G is perfectly general and c_i depends on program attributes which may be tangible or intangible. As an example, we may choose G to be a linear functional. Specifically, we will consider:

$$G = \delta_1 F_1(a_{1i}) + \delta_2 F_2(a_{2i}) + \ldots + \delta_k F_k(a_{ki})$$

and δ_i being either 1 or 0 (15)

for $i = 1, 2, 3, \ldots, n$

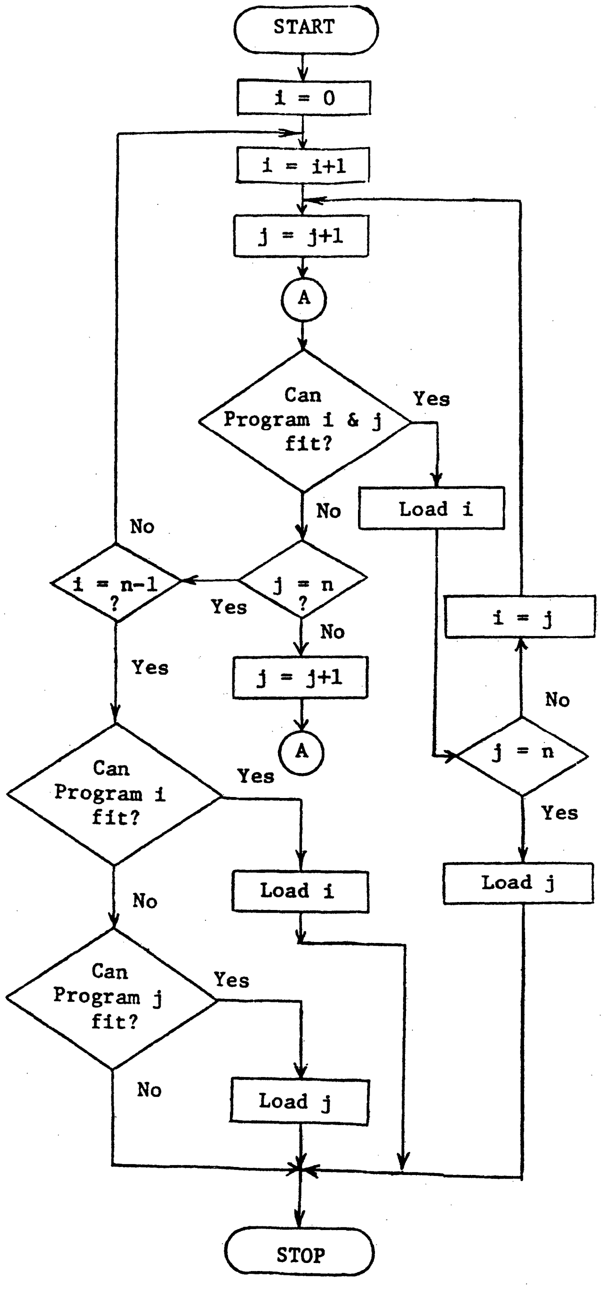


Fig. 1 Flow chart of a heuristic algorithm for finding program loading sequence, for $n \geq 3$.

For the simplest case, suppose we consider the value to be a function of only one parameter. That is:

$$\delta_1 = 1 \quad \delta_j = 0 \quad \text{for } j = 2, \ldots, k$$

$$\text{then} \quad c_i = F_1(a_{1i}) \quad \text{for } i = 1, 2, \ldots, n \qquad (16)$$

If $a_{1i} = \lambda_i$, i.e., the size of the program x_i and if we choose to define the function $F_1(a_{1i})$ as

$$F_1 = \frac{\alpha}{a_{1i}} \qquad (17)$$

where α is a constant, then the generalized value c_i so defined is inversely proportional to the size of program x_i. When we set up our linear objective function as

$$\text{Maximize:} \quad c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$$

we are, in effect, getting a maximum degree of multi-programming as a result. This is easy to see when we realize that the value-to-size ratio in this case is inversely proportional to program size, and a descending order of value/size ratio means an ascending order of the program size. When using largest ratio first algorithm, if this is treated as continuous variable case, or using the sub-optimal algorithm in Figure 1 as in integer case, the programs will be loaded in a sequence with the smallest being the first.

If we include the possibility that each job has been assigned a priority class, it is quite natural to incorporate the priority scheme into our frame-work of the generalized value concept. Consider that a_{1i} being the size of program x_i and $F_1(a_{1i})$ as being defined by (17). We may consider that a_{2i} is the index to a certain priority class to which program x_i belongs.

Furthermore let us define:

$$F_2(a_{2i}) = \beta_i, \quad i = 1, 2, \ldots, n \qquad (18)$$

where β_i is a constant. Then the generalized value c_i is:

$$c_i = F_1(a_{1i}) + F_2(a_{2i}) \quad i = 1, 2, \ldots, n \qquad (19)$$

where:

$$F_1(a_{1i}) = \frac{\alpha}{a_{1i}}$$

$$F_2(a_{2i}) = \beta_i \qquad (20)$$

$$\delta_i = \delta_2 = 1, \quad \delta_j = 0 \quad j = 3, 4, 5, \ldots, k$$

We can see that the priority class is a way to designate a certain "urgency", based on whatevere predetermined guidelines that the system employs, to each individual program. This, by itself, is artificial and the artificially chosen number, β_i, is a reflection of this perception. The choice of α is also somewhat arbitrary, so that the absolute values of both $F_1(a_{1i})$ and $F_2(a_{2i})$ are compatible, e.g., of the same order of magnitude. But this is only one of the possibilities. We can just as easily assign the β_i's to be at least one order of magnitude larger than those of $F_1(a_{1i})$'s.

In so doing, the allocation policy becomes strictly priority oriented. A more balanced approach can be implemented readily by simply adjusting the relative magnitudes between functions F_1 and F_2. In general, we may remark that the generalized value of an individual program is the composite of a set of functions whose parameters may be chosen from intrinsic program properties, such as its size, or subjective reasons, such as priority classes, or both.

Program Execution - Dynamic Planning

In the previous section we have examined the problem of program loading. We have, in fact, treated such action in a static manner. It is static in the sense that our goal is achieved by following an optimizing plan for that particular instance, namely, the instance of the operating system's allocating activity. We have grouped all such activities into "concentrated" points in time. But in a multiprogramming environment, not all programs terminate at the same time. Each time a program (or a particular step of a program) is done, the system either removes this program or continues onto its next job step. If a program is being removed, then memory space it once occupied would be available, thus making it possible for other programs waiting on the system job queue to be loaded, i.e., to be allocated memory space. Consequently, the activities of loading and removing jobs are interspersed throughout the system up time. Even if we may be assumed to have loaded all the programs into memory to the extent possible, the termination, and hence the removal, sequences for programs cannot be predicted, due to the fact that the system ready queue is managed in a dynamic fashion. This dynamic management of the system ready queue constitutes what is considered to be the scheduling activity. In this context, scheduling should not be confused with allocation; one does not necessarily imply the other and vice versa. Furthermore, we have equated scheduling to execution sequences for programs. If we again consider the idealized situation in that all the program loading activities are "concentrated" and that no program will be removed individually until most (maybe all) are completed (terminated), then the two major operating system activities are, in effect, taking place cyclically. If each of such complete cycles is called a period, then we may ask what sort of planning action can we make, i.e., loading and executing, so that an objective function is optimized over several periods? Before supplying answers to this question, we must first decide upon the objectives. Of course, the memory spaces as necessary and scarce resources are central to this question. Now, it can be restated: how can we best utilize a given amount of memory space in a given processing environment? The loading (allocating) and scheduling (executing) activities thus become the means to an end. This brings up a well known model in mathematical programming applications, namely, the Warehousing Model [3]. Essentially, the warehousing model is dealing with the question of, given a fixed capacity and the buying and selling prices of commodities over several periods of time, what action should the warehouse owner take so that his profit is maximized. In the ensuing discussions, we will examine this question in the context of operating systems.

Formulation for the Identical Programs Case

Let us consider a multiprogramming system where all the users (programs) are idnetical in size. This is neither an over simplification nor too far fetched a situation. Many so called express job queues are prime examples of this category, in which every user program is (normally) given a fixed, equal amount of memory space. The user programs are identical to one another in size only, not the amount of computations.

We will denote a list of variables as the following:

Let x_i be the total memory space occupied during period i,
y_i be the total removed memory space during period i,
M be the total memory space,
I be the initial occupied memory in period 1,
d_i be the cost per unit memory during period i,
g_i be the gain per unit memory during period i.

While some of the variables are self-explanatory, others will need further clarifications. The variable x_i is in fact the sum total of all the memory requirements for programs that are loaded in period i and y_i represents the total amount of memory space being freed, due to the termination of programs during period i. The other two variables, d_i and g_i, are artificial quantities. We may think of the memory space as being the necessary resource for certain productive activities and that it incurs a cost when being occupied; and the system accrues profit (gain) when programs are being run to completion and subsequently removed from memory.

We have pointed out earlier that we view this as two major activities taking place in a cyclic manner. To "start" our problem, we must designate one of the two activities as the starting point in the model. Let us therefore assume that, initially, memory is loaded with programs, up to I units. Thus, we have arbitrarily fixed the processing activity to be the beginning activity in period 1; loading activity would follow, thus completing period 1, etc. It can easily be shown that, in general, for i = n, we have the loading constraint as

$$\sum_{i=1}^{n} (x_i - y_i) \leq (M - I), \qquad (21)$$

and the processing constraint as

$$y_i \leq I + \sum_{i=1}^{n-1} (x_i - y_i). \tag{22}$$

From the definitions of d_i and g_i we see that the net gain for each period i would be

$$(g_i y_i - d_i x_i) \tag{23}$$

and it is natural to state our objective over n periods to be

$$\text{Maximize:} \quad \sum_{i=1}^{n} (g_i y_i - d_i x_i). \tag{24}$$

Combining (21) and (22), we can write down the structural matrix as follows:

$$A = \begin{bmatrix} 1 & & & & -1 & & & \\ 1 & 1 & & & -1 & -1 & & \\ \vdots & & \ddots & & \vdots & & \ddots & \\ 1 & 1 & 1 \ldots & 1 & -1 & -1 & -1 \ldots & -1 \\ 0 & & & & 1 & & & \\ -1 & 0 & & & 1 & 1 & & \\ -1 & -1 & 0 & & 1 & 1 & 1 & \\ \vdots & & \ddots & & \vdots & & \ddots & \\ -1 & -1 \ldots & -1 & 0 & 1 & 1 & 1 \ldots & 1 \end{bmatrix} \tag{25}$$

and let λ be the column vector of direct variables and b be the column vector of constraints, i.e.,

$$\lambda = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad b = \begin{bmatrix} M - I \\ M - I \\ \vdots \\ M - I \\ I \\ I \\ \vdots \\ I \end{bmatrix}. \tag{26}$$

Furthermore, if we denote w to be the row vector of dual variables t_i's and u_i's corresponding to x_i's and y_i's, i.e.,

$$w = [t_1 \; t_2 \; \ldots \; t_n \; u_1 \; u_2 \; \ldots \; u_n] \tag{27}$$

then we can write down both the Direct formulation, and its Dual, of the linear programming problem as:

(1) Direct Problem

$$\begin{aligned} &\text{Maximize:} \quad c\lambda \\ &\text{Subject to:} \quad A\lambda \leq b, \quad \lambda \geq 0 \end{aligned} \tag{28}$$

(2) Dual Problem

$$\begin{aligned} &\text{Minimize:} \quad (M - I) \sum_{i=1}^{n} t_i + I \sum_{i=1}^{n} u_i \\ &\text{Subject to:} \quad wA \geq c, \quad w \geq 0 \end{aligned} \tag{29}$$

where c is a row vector, i.e.,

$$c = [-d_1 \; -d_2 \; \ldots \; -d_n \; g_1 \; g_2 \; \ldots \; g_n] \tag{30}$$

Rather than the usual simplex method for solving the Direct Problem, a special algorithm [3] can be employed to attack the Dual Problem instead, due to the nature of this problem as depicted by the special form of its structural matrix shown in Eq. (25).

Example

Let us consider the memory usage question of five periods, with the cost and gain in each period as given in Table 1. The total memory capacity is 200 units and prior to period 1, 100 units of memory space had been occupied.

Table 1 An Example with Five Periods

Period (i)	Cost (d_i)	Gain (g_i)
1	20	25
2	20	35
3	20	21
4	20	40
5	20	50

M = 200 units, I = 100 units

Notice that the costs are identical in every period. Also, it should be clear that these cost and gain figures have no absolute meaning; only their relative magnitudes may reflect upon our system policy.

By using the special algorithm[3], the solution to the minimization problem of Eq. (29) can readily be found to be:

$y_1 = 100,$

$y_i = 200, \quad i = 2, 3, 4, 5;$

$x_i = 200, \quad i = 1, 2, 3, 4,$

$x_5 = 0.$

Thus, the processing-loading pattern as shown in Fig. 2 is optimal in the sense as defined by Eq. (24).

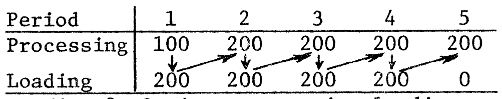


Fig. 2 Optimum processing-loading patterns for five periods

If we modify slightly the values given in Table 1, the resultant "program" might change accordingly. Suppose, g_2 has a value of 17 instead of 35, it can be shown that

$$y_1 = 100,$$

$$y_2 = 0,$$

$$y_i = 200, \quad i = 3, 4, 5;$$

$$x_i = 200, \quad i = 1, 3, 4,$$

$$x_2 = x_5 = 0.$$

This results in an optimum processing-loading pattern as shown in Fig. 3.

Period	1	2	3	4	5
Processing	100	0	200	200	200
Loading	200	0	200	200	0

Fig. 3 Optimum processing-loading pattern with period 2 inactive

Clearly, it can be shown to be true that if for any period i, $i \neq 1$, $d_i > g_i$, then during that period there will be no processing in the final optimum plan. In the extreme case that $d_i > g_i$ for all i, then optimal strategy is simply to process everything already in the memory, i.e., the initial load, and stop. In short, it is no longer advantageous to continue to operate the memory system under such circumstances. Or, viewed differently, the entire productive system (processor and memory) in this case cannot satisfactorily carry out the processing demand according to some predetermined performance criterion.

Multiprograms with Different Sizes

This represents a more typical multiprogramming environment, where different user programs have different sizes. We will, based upon the ideas and results propounded in the immediately prior sections, inquire into some possible scheduling disciplines.

Let us assume that there are m different programs loaded into memory to be processed, each with size I_i units, $\Sigma_i I_i \leq M$. If we partition the memory space exactly according to each I_i, then we may view the system as having m independent memory sections or more, each with a capacity of I_i units, with the possible exception being that portion of the memory space where no program can fit in. Furthermore, each section is full; except the fragmented portion, which is empty. Each individual section can now be viewed as similar to the problem treated previously, but with only one program and such that this program takes up the entire available space.

Now we will redefine the notion of "period". A program's processing can be delayed due either to I/O activity or through timer interrupt. Therefore, the processor is being switched among all the resident programs based on either a cyclic rule or some "dynamic" discipline. If it is cyclic, then it requires no decision on the part of the system, once all the programs are in memory and the system ready queue is thus formed. However, because of the unpredictable nature in terms of timing of I/O activities or due to priority considerations, a program may not always be able to continue even though the processor has made its "round" and back again. In this case, the particular program is being "skipped" for the time being. The definition can now be stated as:

<u>Definition</u> A <u>period</u> for program i is the time between 1) processor entering and leaving (active), or 2) leaving and returning (delayed), or 3) a skip, of program i.

Note that we consider that the processor, after entering a program and upon interrogating the condition, decides not to stay, to be a skip. Note also that periods defined are possibly of unequal length within a program and among programs.

The objective function for the entire system of m programs over n periods can be stated as:

$$\text{Maximize:} \quad \sum_{j=1}^{n} \sum_{i=1}^{n} (g_{ij} - d_{ij}) I_i \qquad (31)$$

where g_{ij} is the gain for program i in period j and d_{ij} is the cost for program i in period j. Clearly, the best (optimal) strategy is that for all i, process those programs for all j such that $g_{ij} \geq d_{ij}$, and skip if otherwise. This is a direct extension of the results discussed in last section. If we follow this approach, then the scheduling discipline is clearly decided upon by the relative magnitudes of g_{ij}'s and d_{ij}'s. Previously, we stated that a particular program may be skipped over, possibly due to some "natural" causes such as waiting for I/O completions. By defining our objective function as Eq. (31) and following the optimal plan, it is possible to exert dynamic control over the scheduling activities by manipulating g_{ij}'s and d_{ij}'s.

Considerations for g_{ij}'s and d_{ij}'s and Scheduling Discipline

We have pointed out earlier that both the <u>gain</u> and the <u>cost</u> of a particular program are something rather intangible and the values chosen to quantify them are indeed artificial. However, artificiality does not imply arbitrariness. We certainly would like to consider the relevant factors in choosing their values so that, in the final analysis, the scheduling disciplines thus resulting constitute viable actions.

(1) $d_{ij} = d_{i1}$ for all j; that is, the cost per unit memory for program i does not change according to period j. Let us further denote that $d_{i1} = d_i$. We would consider this cost as a function of both the program size and memory speed, i.e.,

$$d_i = \phi(I_i, p)$$

where I_i is the size of program i and p is the speed of the memory. On first glance, it would seem redundant to include I_i as a parameter since d_i is already the cost of per unit memory. But, upon closer examination, this definition would give us the freedom to "favor" programs according to their sizes. For example, if we define d_i to be

$$d_i = \frac{\eta I_i}{p} \qquad \eta = \text{constant}$$

then the smaller program will be favored since the cost will be higher for the larger programs. Also, defined as above, d_i is inversely proportional to p, the memory speed, in units of time and the higher the speed, the higer the cost of d_i. There are, of course, many possible choices of relevant parameters and many possible functionals. We only suggest one here so as to illustrate a point.

(2) Recall that in previous sections we have discussed the "loading activity" of the system, based on the concept of generalized value c_i for program i, and we will utilize this value to start the scheduling cycle. Specifically:

i) Let $g_{i1} = c_i$

ii) If at period j, program i is being skipped over, then for period (j + 1), set

$$g_{i(j+1)} = g_{ij} + \gamma d_i \qquad 0 < \gamma \leq 1.$$

This reflects the thinking that every time a program is being skipped, rather than increasing the cost of residency, we instead think of it as being potentially more valuable, i.e., higher gain, to process this program at a possible earlier time. Therefore, we increase its gain per unit memory proportional to its cost for the next period. Hence, the dynamic nature is reflected in the monotonicity of g_{ij} while d_i remains constant.

A possible scheduling discipline is as follows:

In period j, select the job with the highest g_{ij} among all i such that $g_{ij} > d_i$ to be processed.

We will make these remarks regarding this particular scheduling discipline:

i) It is priority influenced since $g_{i1} = c_i$ and c_i is the generalized value for program i which can be directly related to priority classes.

ii) No program will be skipped indefinitely since g_{ij} is monotonically increasing and will be processed eventually. In a way, this is dynamic readjustment on priority while the choice of c_i is static.

iii) The actual schedule depends on the function ϕ and also the constant γ.

Conclusion

We have discussed the ideas of activities and activity aggregates. Unlike the notion that the operating system can be modelled as a set of interacting processes, we view the system as a conglomerate of interdependent activities; interdependent in the sense that they either compete for resources or their action sequences necessarily follow each other. Constrained resource allocation problem and linear objective functions lead to linear programming problem; its mathematical underpinnings are well known. However, by studying the solution process of the programming problems, a more practical algorithm can be established. By presenting a sub-optimal yet practical algorithm in optimizing a general linear objective function, we have in effect, suggested a mechanism for optimization while the generalized value concept provides a way to formulate any policy, e.g., allocation policy, based on some chosen program attributes. The steps, or the mechanics, of an algorithm can be implemented readily in the controlling module(s) within the operating system while the modification of a policy, as for example the allocation policy, can be carried out simply and effectively by changing the "values" associated with individual programs.

References

[1] Dantzig, G. B., "The Programming of Interdependent Activities: Mathematical Model", Activity Analisis of Production and Allocation, T. C. Koopman, ed., New York, John Wiley and Sons, Inc., 1953.

[2] Little, J. D. D., Murty, D. G., Sweeney, D. W., and Karel, C., "An Algorithm for the Travelling Salesman Problem", Operations Research, 1963, 11, pp. 972-989.

[3] Charnes, A. and Cooper, W., "Generalization of the Warehousing Model", Operational Research Quarterly, Vol. 6, No. 4, December 1955, pp. 131-172.

A SCHEME FOR THE PARALLEL EXECUTION OF SEQUENTIAL PROGRAMS[†]

C.V. Ramamoorthy and W.H. Leung
Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California
Berkeley, California 94720

Abstract -- A scheme for the parallel execution of sequential programs is described in this paper. The scheme does not require an extensive parallelism detection procedure before the actual execution of the program. Instead the precedence relation among statements being executed is preserved automatically by the synchronization action of monitors on the processors during execution. The monitoring process is aided by two pieces of information: (1) the reference table which indicates how a variable is used in the program statements, and (2) a stack of trace vectors which keeps track of the execution order of statements.

Introduction

The statements of a sequential program are coded serially as if they are to be executed by a processor one at a time. In this paper, however, we are concerned with the simultaneous execution of more than one of them.

Two types of statements in a sequential program are considered: (1) assignment statements and (2) branch statements. An <u>assignment statement</u> is of the form $X = E_X$ where X is a variable defined by the arithmetic or logical expression E_X. A <u>branch statement</u> is indicated by <u>If</u> c <u>then</u> $(s_1,\ldots,s_n)$ where c is a conditional expression and $s_1,\ldots,s_n$ are statement labels. c can have n possible outcomes. The j-th outcome directs execution to statement s_j. A branch statement can be unconditionally written as GOTO s_j. In a DO-loop the last statement which branches backward to complete the loop can be considered a conditional branch statement.

Existing schemes [BERN 66, RAMA 69, BAER 73, KUCK 75] for the parallel execution of a sequential program require a procedure to detect parallelism in a program before its execution. Statements thus determined to be parallel executable must satisfy the following condition. Let R_i and W_i be the variable space read and updated by statement s_i. If s_i and s_j can be executed at the same time then $A_{ij} = (R_i \cap W_j) \cup (R_j \cap W_i) \cup (W_i \cap W_j) = \emptyset$.

Two observations can be made: (1) the parallel executable statements share limited variable space (only $R_i \cap R_j$ is not required to be empty) and (2) the processors executing them do not communicate with each other. The scheme proposed here relaxes these two restrictions so that statements being executed at the same time can have non-empty A_{ij}'s. The execution will be monitored to preserve the inherent precedence relation among statements. Some static information about the use of variables will be generated before execution to aid the monitoring procedure. But the overhead of doing so will be considerably smaller than an extensive parallelism detection procedure. The monitoring process also requires some dynamic information to trace the execution order of statements.

We use the notion <u>task</u> P_i to represent a statement s_i under execution. Figure 1 depicts the conceptual basis of the scheme. Given two tasks P_i, P_j and their corresponding A_{ij}, procedures called monitors are placed on the access paths of P_i and P_j to A_{ij}. The monitors regulate the execution of P_i and P_j to preserve the precedence between them. The computer model evolved from this idea will have a monitor associated with each processor as shown in Figure 2.

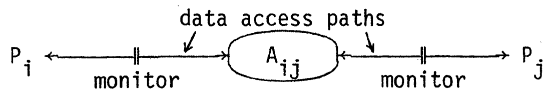


Figure 1. Monitoring parallel execution.

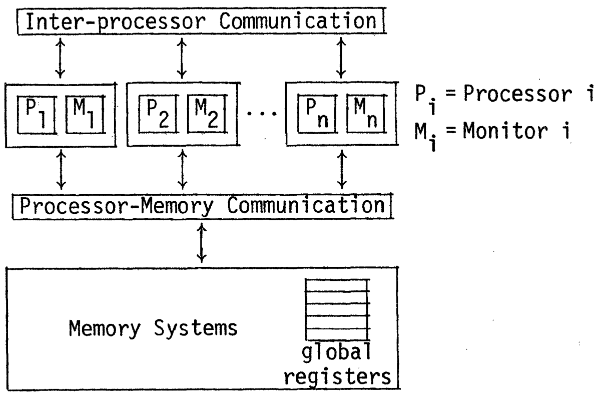


Figure 2. A multi-processor model to carry out the scheme.

Theoretical Basis

A sequential program can be modeled by a direct graph. A node s_i in the graph represents the i-th statement of the program. An edge (i,j) in the graph indicates the flow of control from node s_i to node s_j. Only the node representing a conditional branch statement has more than one

[†]Research sponsored by U.S. Army Research Office Contract DA-ARO-D-31-124-73-G157 and National Science Foundation Grants DCR72-03734-A01 and DCR74-21248.

outgoing edge. Usually an exclusive-or sign ⊕ is attached to it indicating that only one of its edges will be carried out during execution. Figure 3 shows an example program graph.

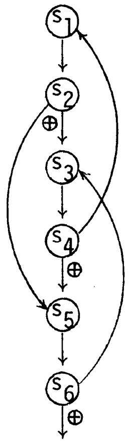


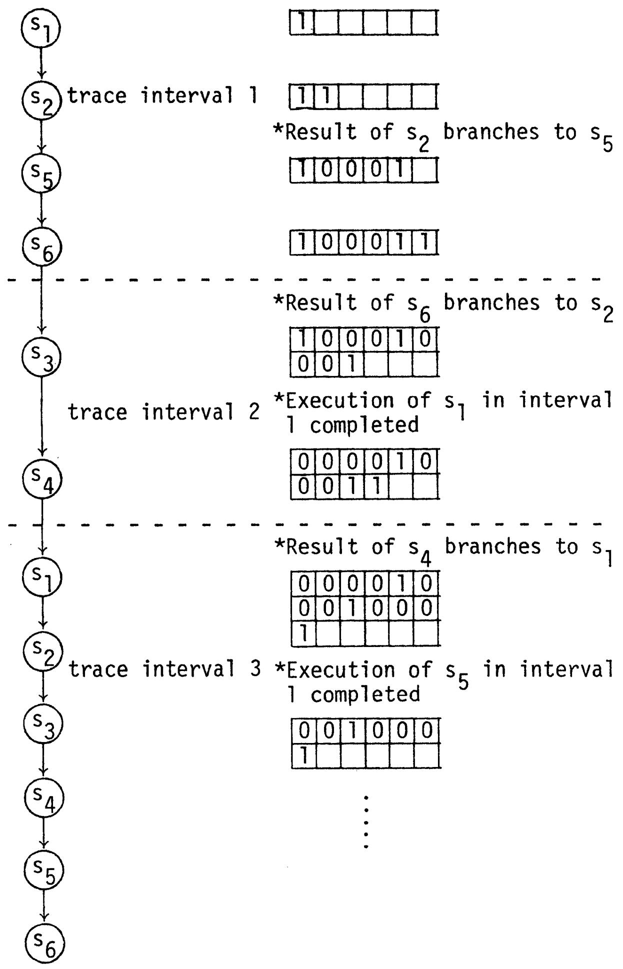
Figure 3. An example program graph.

When a sequential program is being processed by a single processor, the execution follows a route in the program graph. More formally, an <u>execution route</u> is an ordered sequence of nodes (statements) and if node s_i precedes node s_j immediately in the sequence, then there exists an edge (i,j) in the program graph. An execution route of the program graph in Figure 3 is shown in Figure 4a.

We denote the <u>execution order</u> of two nodes s_i and s_j as $s_i \rightarrow s_j$ if s_i is specified by the route to be executed before s_j. Because of conditional branch statements, the execution order of program statements is generally unknown before the execution of the program. However, if given a program block which consists of only assignment statements, then $s_i \rightarrow s_j$ if and only if $s_i < s_j$.

The precedence which must be preserved among statements is related to both their execution order and data dependency. Given two statements s_i and s_j, if (1) $s_i \rightarrow s_j$, (2) $(W_i \cap R_j) \neq \emptyset$ and (3) for any s_k such that $s_i \rightarrow s_k \rightarrow s_j$ and $W_k \cap (W_i \cap R_j) = \emptyset$, then we say s_j is an <u>immediate data dependent</u> of s_i. A variable in $(W_i \cap R_j)$ is <u>read correctly</u> by P_j (the task corresponding to statement s_j) if it is not modified between the time it is updated by P_i and the time it is read by P_j. We assume that a statement s_j is <u>executed correctly</u> if all variables in R_j are read correctly. It follows that a program will be executed correctly if all the statements in the execution route are executed correctly. This is what this scheme must guarantee.

For a statement s_j to be an immediate data dependent of s_i, it implies that P_j should not read variables in $(W_i \cap R_j)$ before P_i updates them. But in order to execute P_j correctly, we must also protect the integrity of the variables in $(W_i \cap R_j)$. For this purpose, we define the



(a) An example execution route for the program graph in Figure 3.

(b) A possible sequence of trace vectors corresponding to the execution route in (a).

Figure 4. Execution route, trace intervals and trace vectors.

<u>direct precedence relation</u> between two statements as the following. Given two statements s_i and s_j, if $s_i \rightarrow s_j$ and $A_{ij} = (R_i \cap W_j) \cup (R_j \cap W_i) \cup (W_i \cap W_j) \neq \emptyset$ then s_i directly precedes s_j with respect to elements in A_{ij}. This means that P_i must finish processing elements in A_{ij} before P_j can operate on them. A_{ij} is called the <u>shared variable set</u>. Figure 6 lists the shared variable sets of the assignment statements in Figure 5.

It should be noticed that given two statements which do not directly precede one another does not mean that they can be executed in parallel. Consider the sequence of statements in Figure 6. s_1 must be executed before s_5 although A_{15} is empty.

However, the direct precedence is important because of the following fact: If the direct precedence relation among all statements along

s_1: A(I,J) = 1.0
s_2: X = X+I
s_3: I = I-1
s_4: J = J-1
s_5: Y(I) = I+X
s_6: Z(I,J) = I*J

Figure 5
An example assignment statement block.

$A_{12} = \emptyset$; $A_{13} = \{I\}$; $A_{14} = \{J\}$;
$A_{15} = \emptyset$; $A_{16} = \emptyset$; $A_{23} = \{I\}$;
$A_{24} = \emptyset$; $A_{25} = \{X\}$; $A_{26} = \emptyset$;
$A_{34} = \emptyset$; $A_{35} = \{I\}$; $A_{36} = \{I\}$;
$A_{45} = \emptyset$; $A_{46} = \{J\}$; $A_{56} = \emptyset$.

Figure 6. Shared variable sets for the example in Figure 5.

the execution route are obeyed, then the statements will be executed correctly in the sense as stated earlier.

The proof is in the following. Let s_j be an immediate data dependent of s_i. It follows that s_i also directly precedes s_j.

Suppose there exists a statement s_k which could modify variables in $(W_i \cap R_j)$ then $W_k \cap (W_i \cap R_j) \neq \emptyset$. Therefore s_k is not a statement between s_i and s_j in the execution route.

If $s_k \rightarrow s_i$ then s_k directly precedes s_i because $W_k \cap W_i \neq \emptyset$. Furthermore, if $s_j \rightarrow s_k$ then s_j directly precedes s_k because $W_k \cap R_j \neq \emptyset$.

If the direct precedence relation among s_i, s_j and s_k are obeyed then P_j will read variables in $(W_i \cap R_j)$ correctly. Since s_i is arbitrary, it follows that P_j can read R_j correctly if the direct precedence relation among all tasks are preserved. Hence s_j can be executed correctly. This will hold for all other statements.

The above argument suggests that if we can (1) determine the execution order of statements and (2) obtain the shared variable sets of statements, then we can preserve the direct precedence relations and execute a sequential program correctly. The monitors in Figure 1 are intended to preserve the direct precedence relation among the statements when they are being executed.

The shared variable sets can be more conveniently represented by a variable reference table which will be described in the following section. A stack of trace vectors can be used to indicate the execution order of statements. It will be shown after the description of the reference table.

Reference Table

It is possible to obtain the shared variable sets for every pair of program statements by preprocessing procedures similar to data flow analysis [ALLE 76]. However, if a program has n statements, then there can be as many as n(n-1)/2 shared variable sets. It will be difficult to manage this large number of shared variable sets in execution time.

One method is to use the reference table in lieu of the shared variable sets. Essentially, the reference table indicates in which statement a variable is referenced. It is commonly used as a debugging aid and many existing FORTRAN compilers can generate it with a small overhead. In our particular implementation, it also shows how the variable is used in a statement. An element in the table is denoted as $RT(X,s_i)$ where X is a variable name and s_i is a statement label. $RT(X,s_i)$ can have one of the following four values:

$$RT(X,s_i) = \begin{cases} 00 & \text{if } X \text{ is not referenced in } s_i \\ 01 & \text{if } X \text{ is read in } s_i \\ 10 & \text{if } X \text{ is updated in } s_i \\ 11 & \text{if } X \text{ is read and updated in } s_i \end{cases}$$

We assume that the table will be searched associatively using a variable name. Therefore the first index of the array RT is a variable name. The reference table for the block of assignment statements in Figure 6 is given in Figure 7.

	s_1	s_2	s_3	s_4	s_5	s_6
A	10	00	00	00	00	00
I	01	00	11	00	01	01
J	01	01	00	11	00	01
X	00	11	00	00	01	00
Y	00	00	00	00	10	00
Z	00	00	00	00	00	10

Figure 7. The reference table for the program block shown in Figure 5.

The reference table is important because it actually indicates the membership of a variable in shared variable sets. It is quite clear from Figure 7 that the variable J belongs to A_{14}, A_{24} and A_{46}. The dimension of a reference table is directly related to the size of a program block. For large programs, it is necessary to partition the program into program blocks and associate a reference table with each of them.

Next, we describe the trace vectors.

Trace Vectors

The trace vectors convey two messages: (1) They show whether a statement in the execution route has been completed or not. (2) They reveal the execution order of statements.

An element of the trace vector is denoted as $TV(u,s_i)$ where s_i is a statement label and u indicates a trace interval. A new trace interval is added when a backward branch is in effect. The element $TV(u,s_i) = 1$ means that s_i is being executed in interval u. $TV(u,s_i) = 0$ implies the execution of s_i is completed or s_i does not appear in the execution route in interval u. Two non-zero trace vector elements $TV(u,s_i)$ and $TV(v,s_j)$ show the execution order between s_i and s_j according to the following:

(1) If $v < u$, then $s_j \rightarrow s_i$.

(2) If $v = u$ and $s_i < s_j$ then $s_i \rightarrow s_j$ otherwise if $s_j < s_i$ then $s_j \rightarrow s_i$.

(3) If $v > u$ then $s_i \rightarrow s_j$.

The trace vectors will be updated frequently during execution to keep track of the execution order of statements. We consider three cases in updating the trace vectors.

(1) When a statement s_i is fetched for execution, the corresponding $TV(u,s_i)$ will be set to one. There is no change in trace interval.

(2) When a statement s_m is fetched as a result of forward branching from statement s_ℓ, the statements between s_ℓ and s_m are not covered in the execution route. Hence, for all $s_\ell < s_i < s_m$, $TV(u,s_i)$ will be reset to zero and $TV(u,s_m) = 1$. There is no change in trace interval.

(3) As shown in Figure 4a, a backward branch from statement s_m to s_ℓ means that some statement s_k, such that $s_\ell < s_k < s_m$, may precede s_ℓ in execution order. To account for this fact, a new trace interval u+1 will be created when s_ℓ is fetched for execution. Suppose the program block under consideration contains n statements. Then $TV(u,s_i) = 0$ for all $s_m < s_i < s_n$ and $TV(u+1,s_j) = 0$ for all $s_i \leq s_j < s_\ell$, since these statements are not covered in the execution route. But $TV(u+1,s_\ell)$ will be set to one.

When the execution of a statement is completed, the corresponding trace vector element will be reset to zero.

Figure 4b shows a possible sequence of trace vectors for the execution route shown in Figure 4a. It should be noticed that before the trace vectors are updated due to the fetching of statement s_i, no task which executes statement s_j and $s_i \rightarrow s_j$ can use the trace vectors.

Statements in different trace intervals can be executed at the same time. We say a trace interval u is used if $TV(u,s_i) = 1$ for some s_i otherwise it is not used. Since at least one processor is executing a statement in a used trace interval, the number of used trace intervals is bounded by the number of processors.

Monitor

It is clear that given the information in the reference table and the trace vector, the direct precedence relations among statements can be determined. We are now in a position to describe the monitors.

A monitor regulates the execution of a processor to preserve the direct precedence relation among statements. Suppose a variable X is referenced by a processor executing statement s_i. If it is found by the monitor that there exists at least one statement s_j which directly precedes s_i with respect to X and has not been completed, then the monitor will issue a message "wait(X)" to interrupt the execution of the processor. On the other hand, if the operation on X has been completed, the monitor sends a "signal(X)" message to inform other monitors in the system.

To match up the speed of the processor, a monitor should be implemented by hardware. But for simplicity and clarity, we shall describe it as a procedure.

A monitor consists of (1) a set of data, the reference table and the trace vectors, which are shared by all other monitors and (2) two procedures which regulate the execution. If variable X is referenced in statement s_i in trace interval u, then depending on the operation (read or update) the monitor will do one of the following procedures:

```
Procedure Monitor Read (variable X; statement
    $s_i$; trace interval u);
  Begin
    For ((all intervals $v < u$ and all state-
      ments $s_j$) or (interval $v = u$ and all
      statements $s_j < s_i$)) DO
      If any ($TV(v,s_j) = 1$ and ($RT(X,s_j) = 10$
        or $RT(X,s_j) = 11$))
        then wait(X)
        else begin
               read(X);
               signal(X);
             end;
  End.

Procedure Monitor Write (variable X; statement
    $s_i$; trace interval u);
  Begin
    For ((all intervals $v < u$ and all state-
      ments $s_j$) or (interval $v = u$ and all
      statements $s_j < s_i$)) DO
      If any ($TV(v,s_j)$ and $RT(X,s_j) \neq 00$)
        then wait(X)
        else begin
               update(X);
               signal(X);
             end;
  End.
```

The proof that these two procedures correctly preserve the direct precedence among statements is quite straightforward and we shall not present it here.

The decision of whether to issue "wait(X)" in either procedure essentially follows two steps: (1) fetch the row $RT(X,s_j)$ from the reference table, and (2) compare $RT(X,s_j)$ with the trace vector to reach the decision.

Figure 8 illustrates the performance of the scheme if the program block in Figure 6 is executed by three processors. There are a variety of methods to improve the performance of the monitor.

But we shall not cover them here.

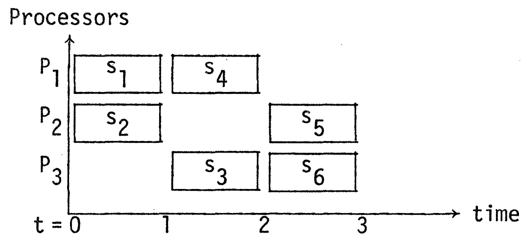


Figure 8. Time chart of executing the program block given in Figure 5.

Final Remark

As a final remark, the scheme has an advantage that it requires little preprocessing overhead since the major part of the parallelism detection is done during execution time. But because of the same reason, it cannot guarantee an optimal scheduling strategy in the sense that the processors are fully utilized. Further investigations are needed to evaluate its effectiveness.

References

[ALLE 76] F.E. Allen, and J. Cocke, "A Program Data Flow Analysis Procedure," Comm. ACM (March 1976).

[BAER 73] J.L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," ACM Computing Survey (January 1973).

[BERN 66] A.J. Bernstein, "Analysis of Programs for Parallel Processing," IEEE TC (October 1966).

[KUCK 75] D.J. Kuck, "Parallel Processor Architecture--A Survey," 1975 Sagamore Conference Proceedings.

[RAMA 69] C.V. Ramamoorthy, and M.J. Gonzalez, "A Survey of the Techniques for Recognizing Parallel Processable Streams in Computer Programs," AFIPS FJCC (1969).

ON FURTHER APPLICATIONS OF THE HU ALGORITHM TO SCHEDULING PROBLEMS

Edgar Nett
Gesellschaft für Mathematik
und Datenverarbeitung
5205 St. Augustin, West Germany

Abstract-- This paper concerns the generation of optimal task schedules for multiprocessor systems. So far, non-exhaustive algorithms for the generation of optimal schedules have been devised only under restrictive assumptions. One of them is the so-called Hu algorithm, which, because of its simplicity, appears to be very attractive for practical applications and, therefore, deserves further investigations. A new solution for the two-processor scheduling problem is proposed which predominantly employs the Hu algorithm. Furthermore, special task dependency structures are introduced which essentially are composed of trees and anti-trees, and include the structure of nested DO-loops. If two additional constraints are imposed on these structures, then the application of the Hu algorithm yields optimal schedules for an arbitrary number of processors.

I. Introduction

The allocation of processors to tasks in a multiprocessor environment has proved to be a problem of high complexity if a high degree of utilization of the available processing power or, which is essentially the same, the minimization of the overall job run time is to be achieved. To process any two tasks simultaneously, it must be made sure that both are independent of each other, i.e. one task must not produce code or data that are required to process the other task, and at no time during their execution must both use identical resources unless, as in the case of reentrant code, it is explicity allowed. Both criteria usually are quite naturally met by two different user programs or by a user and a system program. Therefore, the simultaneous processing of a independent program is commonly preferred to simplify the job of the task dispatcher.

However, in some cases it might be desirable or even necessary to significantly reduce the run time of a particular program by executing independent tasks within the program simultaneously. Tasks of this sort may be single instructions or small instruction sequences such as, for instance, the branches of a DO-loop. An optimal processor allocation at this level usually is very difficult to accomplish for there may be exist rather complex and strong dependencies between the tasks that must be strictly obeyed during program execution in order to produce correct results.

The task dependencies within a program which are commonly caused by data transfers from one to another, can be formally described by a tuple $(T,\lessdot)$, where $T = (t_1,\ldots,t_n)$ is the set of tasks and $\lessdot$ is a partial ordering on T. If $t_i \lessdot t_j$, then, the execution of t_j must not be initiated before the execution of t_i has been completed. If, however, $t_i \not\lessdot t_j$ and $t_j \not\lessdot t_i$, then both tasks are said to be independent of each other and, therefore, may be executed simultaneously.

The tuple $(T,\lessdot)$ can be represented as a task graph $G = (T,\lessdot,a,e)$, in which the tasks are shown as nodes, where a and e are the single entry and exit nodes, respectively. $\lessdot$ is represented by directed edges so that there is a directed edge from task t_i to task t_j if and only if $t_i \lessdot t_j$ and there is no task t_k so that $t_i \lessdot t_k \lessdot t_j$. The task scheduling problem can be considerably simplified if the nodes in a task graph G are considered as task units each of which requires one unit of time for execution. This unit of time is to be chosen so that a real task can be represented as a chain of task units in the corresponding task graph. In the following, the term 'task' always refers to a task unit. Furthermore, it is assumed that the scheduling is non-preemptive and that a processor is left idle for a period of time only if no task is executable within this period (demand scheduling).

If an optimal task scheduling strategy is to be performed, then, usually it does not suffice to simply assign an idle processor to any one task out of a pool of executable tasks, i.e. tasks whose predecessors in the task graph have already been processed. In addition, some order of priority among the executable tasks must be established that can be derived from the structure of the task graph.

One of the simplest priority criteria is the so-called level criterion. The level $\ell(t)$ of a task t is defined as the maximum number of tasks that can be encountered on a path from task t to the exit task e in the corresponding task graph. The level criterion gives priority of execution to the tasks with the highest levels among all tasks which, at some instant of time, are executable.

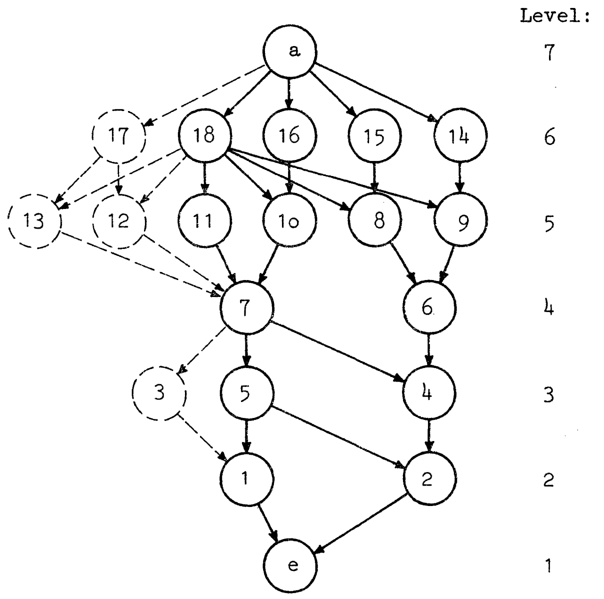
So far, Hu [2] has demonstrated that a scheduling algorithm which exclusively applies the level criterion suffices to generate optimal schedules for partial task orderings that feature a tree structure. This structure, however, does not represent task dependencies which typically can be found in conventional computer programs.

This paper is to present two more scheduling problems for which the Hu algorithm generates optimal schedules. In section II, it will be shown that the classical two-processor problem, for which Coffman/Graham have proposed a sophisticated labelling scheme [1], can be solved using predominantly the simpler level criterion, and in section III, a special task dependency structure, essentially composed of trees and anti-trees, is introduced for which the Hu algorithm yields an optimal schedule as well.

II. The two-processor problem

It appears useful to set out with an informal discussion of the problems that arise when applying the Hu algorithm to task scheduling in a two-processor system.

Fig.1 shows, as an example, a typical task dependency structure. In this graph, tasks (nodes) of identical level are arranged in horizontal rows; the levels are identified to the right of each row.



Optimal Hu-schedule S_o .

	1	2	3	4	5	6	7	8	9	10	11
P_1	a	18	15	14	11	1o	9	6	3	2	e
P_2	φ	16	17	12	13	8	7	5	4	1	φ

Fig.1. Example of a task graph and an optimal schedule for it. The numbers in the nodes serve as task identifiers.

Consider first a reduced graph from which the nodes 17, 13, 12, 3 and the corresponding directed edges are removed. This graph features an even number of tasks in every level, except those levels which only consist of the entry task a and the exit task e, respectively. Hence, after execution of the entry task a, all tasks having an identical level can be processed pairwise concurrently in arbitrary order, so that all tasks having different levels can be processed sequentially in the order of monotonically descending levels until task e is set free. The resulting schedule must be optimal since in all but the first and last time slot both processors are busy.

A more complex situation comes about, if the tasks 17, 13, 12, 3 are included in the task dependency structure of Fig.1. Now, the level 6 comprises an odd number of tasks which become free for execution after task a has been completed. To achieve an optimal schedule, one of the tasks of this level must necessarily be processed concurrently with a task of level 5. This task, however, cannot be selected arbitrarily. As can be recognized in Fig.1, task 18 is succeeded by all tasks of level 5 and, therefore, definitely cannot be processed together with a task of level 5 and, therefore, not as the last task of level 6 either.

A similar situation occurs when processing the tasks of level 5. Since one of these tasks has to be executed together with a task from level 6, again, an odd number of tasks is left over. Consequently, one of them must be performed concurrently with a task from level 4. If one of the tasks 10, 11, 12, 13 from level 5 is chosen, then it can only be executed together with task 6 of level 4. Thus, only task 7 is left over for execution in the next time slot, for it blocks all tasks of lower levels. If, however, task 8 or 9 of level 5 is paired with task 7 of level 4, then task 6 can be paired, for instance, with task 3 of level 3, which leaves over an even number of tasks in the remainder of the levels 3 and 2. Intuitively, it appears that these situations can only be resolved if the sets of successors of the tasks are taken into consideration. Whenever the number of executable tasks of the highest level becomes odd, then it seems necessary to select the task with the smallest number of successors for execution together with a task of the next lower level.

Thus, the following extended Hu algorithm is proposed for task scheduling in a two-processor system:
In every time slot, the tasks with the highest levels among the executable tasks are assigned to the processors. If there is a tie among more than two tasks and the number of them is even, then an arbitrary pair of these tasks can be selected. In the case that their number is odd the task with the smallest number of successors has to be processed last.

To verify that this algorithm is optimal, the notions of 'dominance' and 'Incompletely Occupied Time Slot (abr. IOTS) are introduced.

A task r is said to dominate a task s in a task graph G if the set of successors of s, N(s), is a subset of the corresponding set N(r) of r.

Two task sets $I = \{t_1,\dots,t_k\}$ and $J = \{s_1,\dots,s_k\}$ are said to have identical structure if there exists an edge (t_u,t_v) in the corresponding task graph G if and only if there exists an edge (s_u,s_v) in G, too. Now we extend the notion of dominance as follows:
I is said to dominate J if and only if for every integer i out of $\{1,\dots,k\}$ it is
$N(t_i) \Rightarrow (N(s_i) - J)$.

These definitions of dominance are slight modifications of those given by Ramamoorthy et. al. [3].

The dominance criterion requires that a task is always executed before or at the same time as those tasks it dominates. We immediately conclude from these definitions:

Lemma II.1: If a task r dominates a task s, then the level $\ell(r)$ of r is greater than or equal to the level $\ell(s)$ of s.

Let us now introduce the notion of an 'IOTS'. Suppose, S is a schedule over a task graph G, then, M_i is defined as a subset of all tasks $t \in T$ that are ready to be executed at the beginning of time slot i, S_i is defined as the subset of all tasks $t \in T$ that are executed during time slot i.

Let $\omega(S)$ be the number of time slots required to execute G according to the schedule S and let i $(1 \leq i \leq \omega(S))$ be a time slot so that $|S_i|$ is smaller than m, the number of processors. Then, i is said to be an IOTS. The following property of an IOTS can be readily verfied:

Theorem II.1: Let i be an IOTS, then the set $N(S_i)$ of all sucessors of the tasks belonging to S_i is equal to the set of all those tasks that have not been executed at the end of time slot i.

To find out whether a schedule S over G is optimal or not we must study the IOTS's of S. However, many of them have no influence on the optimality of S. (See for example the first and last time slot of the schedule in Fig.1.)

An IOTS i is called irreducible if and only if there is at least one optimal schedule R over G so that $R_i = S_i$, otherwise, i is called reducible. Henceforth, we are only interested in the reducible IOTS's, for which from theorem II.1 we can derive the following important property:

Corollary II.1: Let i be a reducible IOTS in a non-optimal schedule S over G. Then, there exists at least one task $t \in S_i$ which in every optimal schedule R over G is executed in a preceding time slot.

This proposition means that there must exist a time slot j which precedes i and at least one task s in the corresponding set S_j so that, in order to generate an optimal schedule, task t must be executed before task s.

Now we can propose the following theorem:

Theorem II.2: Let S be a Hu-schedule for two processors over some task graph G and let the dominance criterion not be violated in S. Then, S is optimal.

Proof: See appendix

We are now in a position to verify that the proposed algorithm is optimal.From Lemma II.1 follows that it suffices to ensure that the dominance criterion is not violated if we have to select for execution two out of more than two tasks which have the same level. A violation has no influence on the optimality of the schedule S if the number of tasks having the same level is even. In this case no processor is idle during the execution of tasks having the same level and the task graph which is left over after execution of all tasks of a particular level is independent of the sequence of execution.If,however, the number of tasks of a level is odd,then we only have to make sure that a dominating task is not executed as the last one. This, however, is in compliance with the proposed algorithm which requires that the task with the smallest number of successors is executed as the last one. Hence, the extended Hu algorithm produces optimal schedules for two-processor systems.

In comparison to the labelling scheme developed by Coffman/Graham [1], the extended Hu algorithm, on the average, requires less computation to produce an optimal schedule. The task priority assignment by the level criterion is to be supplemented by the dominance criterion, i.e. by an assessment of the number of successors of a task, only if the number of executable tasks of the highest level is odd. In contrast to this, the Coffman/Graham algorithm assigns a unique label, which implicitly reflects the number of successors, and thereby a unique priority to every task of the same level even though this further distinction of priorities may not be necessary.

III. Scheduling of tree-antitree task dependency structures

As has been pointed out in the introduction, optimal task scheduling in a multiprocessor system is known to require exhaustive algorithms if the number of processors exceeds two and if arbitrary partial task orderings (dependency structures) are permitted. Then, the complexity of the algorithms is polynomial complete [5].

Obviously, this complexity can be reduced to yield nonexhaustive scheduling algorithms only by two measures, of which one, the restriction on the number of processors leads to a simpler algorithm only in the two-processor case. The other, a simplification of the permissible task dependencies has been successfully applied only to tree and antitree structures, for which the Hu algorithm was originally developed.

Intuitively, it appears very promising to follow this direction and to try to consider simple task dependency structures which can typically be found in computer programs and are easy to schedule. Fortunately, it turns out that (nested) DO-loops not only are rather simply structured but also provide the overwhelming majority of concurrently executable tasks (instructions) within a program.[4] Consider, as an example, the following ALGOL program which generates a symmetrical matrix.

```
Proc SYMM (n,a)
  integer n; array a(1:n, 1:n);
  begin integer  i, j, k;
        for  i : = 1  step  1  until  n  do
             a(i,i) : = 0;
        for  i : = 1  step  1  until  n-1  do
             begin
                   k : = i+1;
                   for  j:=k step 1 until n do
                        begin
                             a(i,j):=.i+j;
                             a(j,i):= a(i,j);
                        end
             end
  end
```

If n = 4 is assumed, then the dependencies between the individual tasks (instructions) are as shown in Fig. 2.

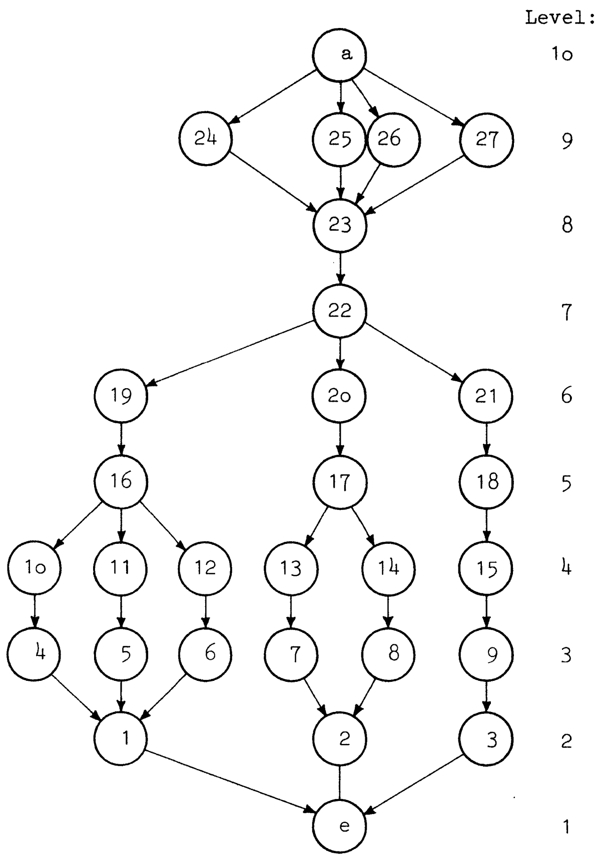


Fig.2: The computational structure of the ALGOL program

The tasks have to be interpreted as follows:

Task a - - - initialization of the first DO-loop

Task 24 up to 27- - - the matrix elements lying on the diagonal are set to 0.

Task 23- - - termination of the first DO-loop

Task 22- - - initialization of the second DO-loop

Task 19 up to 21- - - integer k is set to 2,3,4, respectively.

Task 16 up to 18- - - initializations of the third DO-loop

Task 4
up to 15- - - computation of the matrix elements
Task 1
up to 3- - - terminations of the third DO-loop
Task e- - - termination of the second DO-loop

If linear substructures are considered as trivial trees, then this kind of structure appears to be composed entirely of trees, the branches of which are joined again by equivalent anti-trees. This, in turn, suggests that optimal task schedules for more than two processors may be generated by exclusively applying the Hu algorithm.

To investigate this problem, first the set of task graphs $\mathcal{G}^*$ which includes the structures in question is informally defined. Graphs of the set $\mathcal{G}^*$ can be constructed recursively out of two basic graph elements by systematically substituting tasks in some task graph $G \in \mathcal{G}^*$ by these elements. These task elements are:

1) graphs consisting of two nodes that are connected by a directed edge;
2) all graphs in which every node i except the entry node a and the exit node e is immediate successor of a and immediate predecessor of e.

   (These graph elements are illustrated in Fig. 3)

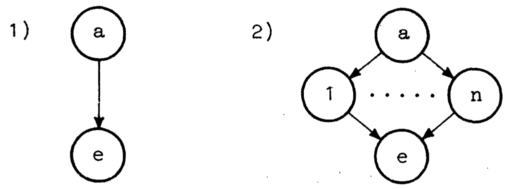


Fig.3: The basic graph elements of $\mathcal{G}^*$

It is important to note that the construction of graphs $G \in \mathcal{G}^*$ must always start with an element of type 2 whose entry and exit node, however, must never be substituted. If two additional conditions are imposed on the graphs from $\mathcal{G}^*$ concerning the number of branches which emanate from the entry node a and the number of tasks in each of this branches, then it can be shown that the Hu algorithm yields optimal schedules.

To show this, the following lemma proves very helpful:

<u>Lemma III.1:</u> Let $G \in \mathcal{G}^*$ be a task graph and let D(a) be the set of immediate successors of the entry task $a \in G$, then $\bigcap_{t \in D(a)} N(t) = \{e\}$, where N(t) is the set of successors of t.

The proof of this lemma follows straightforwardly from the structure of the task graphs from $\mathcal{G}^*$.

Now, the following theorem can be formulated:

<u>Theorem III.1:</u> Let $G = (T,<,a,e) \in \mathcal{G}^*$, let $Y = (Y_1,\dots,Y_p)$ be a partition class on D(a) that meets the following conditions:

1) $|N(Y_i) \cup Y_i| \geq L-2$ $(1 \leq i \leq p)$, where L is the length of the longest path in G and $N(Y_i) := \bigcup_{t_i \in Y_i} N(t_i)$
2) $p \geq m$

Then, every Hu-schedule over G is optimal.

<u>Proof:</u> see appendix

The conditions of this theorem essentially require that the number of branches emanating from the entry node a is at least equal to the number of processors, and that the number of tasks in every branch must exceed a certain limit which is roughly given by the largest number of task levels in a single branch.

These two conditions can easily be met by (nested) DO-loops, if the number of branches of the outermost loop is greater or equal to the number of processors.

Consider as an example, again, the task graph of Fig.2. It is composed of two subgraphs from the set $\mathcal{G}^*$, the first one extends from the node a to the node 23, and the second one extends from the node 22 to the node e. To generate an optimal schedule for the entire task graph structure, the optimale schedules of the two subgraphs may simply be concatenated.

Since in each of the two subgraphs all branches emanating from the respective entry nodes comprise the same number of task levels, (L=3 in the upper subgraph and L=7 in the lower subgraph), the first condition of Theorem III.1 is fulfilled. The second condition is met if the number of processors, for instance, is assumed to be three. This is equal or smaller than the number of branches in both subgraphs. Hence, an optimal schedule for three processors generated by the Hu algorithm is of the following form:

Optimal Hu-schedule

P_1	a	24	27	23	22	21	16	13	10	4	7	1	e
P_2	ϕ	25	ϕ	ϕ	ϕ	20	17	14	11	5	8	2	ϕ
P_3	ϕ	26	ϕ	ϕ	ϕ	19	18	15	12	6	9	3	ϕ
	1. Subgraph				2. Subgraph								

Concluding remarks

The Hu algorithm seems to be most suitable for practical scheduling applications, because it is relatively simple in comparison to other algorithms.

In this paper an extension of the Hu algorithm is applied to the two-processor scheduling problem and to the scheduling of special task structures that include nested DO-loops.

Acknowledgements

The author would like to thank Dr. W. Kluge for his help and guidance with this research, and he would also like to thank his colleagues for their useful comments and helpful suggestions.

References

1) Coffman, E.G. and Graham, R.L.:
'Optimal Scheduling for TWo-Processor Systems', Acta Informatica, Vol. 1, No. 3, 1972

2) Hu, T.C.:
'Parallel Sequencing and Assembly Line Problems' Op. Res., Vol.9, No.6, Nov. 1961

3) Ramamoorthy, C.V. et. al.:
'Optimal Scheduling Strategies in a Multiprocessor System', IEEE Trans. on Comp., Vol. C-21, No.2, Febr. 1972

4) Baer, J.L.:
'A Survey of some Theoretical Aspects of Multiprocessing', Computing Surveys, Vol.5, No. 1, March 1973

5) Ullman, J.D.:
'Polynomial Complete Scheduling Problems', Techn. Report 3, Dept. of Comp. Science, Univ. of California at Berkeley, March 1973

Appendix

The correctness of Theorem II.2 is shown by proving its contraposition.

From the definition of a reducible IOTS it follows that in a non-optimal schedule there are two reducible IOTS's. As can be derived from Corollary II.1, a reducible IOTS comes about if, prior to this IOTS some task s has been executed before some other task r, although task r must be executed before task s in order to obtain an optimal schedule.

It will be shown that then task r dominates task s.

Proof of theorem II.2: Let S be a non-optimal Hu-schedule, let i be the reducible IOTS with the smallest number and let $S_i = \{r_o\}$. Then, according to Corollary II.1 there must be a time slot $j < i$ with $S_j = \{s_o, t\}$, in which r_o should be executed instead of s_o to make the schedule optimal. This, in turn, implies that s_o must be executed later than r_o, and, hence, that $s_o \notin V(r_o)$ and, furthermore, $N(s_o) \not\supseteq N(r_o)$, i.e. for the levels of s_o and r_o must hold:

I) $l(s_o) \leq l(r_o)$

Let q be the greatest integer so that in S there exists a task $r_1 \in S_q$ with $r_1 \in V(r_o)$. We now have to distinguish between the following three cases:

Case 1): $j < q$

Then, the tasks s_o and t are executed before task r_1 and, therefore, we can state: $r_1 \neq t$. Since r_1 is a predecessor of r_o, $l(r_o)$ is smaller than $l(r_1)$ and, since $l(s_o) \leq l(r_o)$, it follows also $l(s_o) < l(r_1)$. Therefore, task t must be a predecessor of r_1, since otherwise r_1 should be executed instead of s_o in time slot j. Consequently, $l(r_1)$ is smaller than $l(t)$. Now we can construct the following chain of relations: $l(s_o) \leq l(r_o) < l(r_1) < l(t)$ and, therefore, $l(t) \geq l(s_o) +2$. However, since the level criterion has been applied to generate the schedule, no time slot $j' < j$ contains predecessors of the tasks t and s_o which have the same level. Hence, there is no possibility to execute task t, and, subsequently, task r_o before task s_o. According to corollary II.1, this is a contradiction to the assumption that i is an reducible IOTS.

Case 2): $j > q$

Then, the tasks s_o and t are executed after the task r_1 and, therefore, t cannot be a predecessor of r_o. Hence, r_o is already executable in time slot j. Since $N(s_o) \subsetneqq N(r_o)$, this means, that r_o dominates s_o at the beginning of the j-th time slot, i.e. the schedule S violates the dominance criterion.

Case 3): $j = q$

In this case, it is $r_1 = t$. Then, by Corollary II.1 there exists an integer $h < j$ with r, $s \in M_h$, i.e. the tasks r and s are executable in the time slot h, where r and s have the following properties:

i) $l(r) = l(s)$

ii) $s \in V(s_o)$; $r \in V(r_1) \cup \{r_1\}$

iii) $s \in S_h$; $r \notin S_h$, i.e. s was executed in time slot h, but not r.

Let h be the greatest integer with these properties, and let us assume that r does not dominate s.

Then, from $l(r) = l(s)$ and $N(s_o) \subsetneqq N(r_o)$ follows that there is a task $p \in (N(s) - N(s_o))$ so that the number of immediate sucessors of p, $|D(p)|$, is greater than 2 because otherwise it is instantly clear that the task set

$I = (N(s) \cup \{s\} - N(s_o))$ is dominated by a subset

$J = (N(r) \cup \{r\} - N(r_o))$ since $N(s_o) \subsetneqq N(r_o)$.

Let $p_1, p_2 \in D(p)$ and, without restriction of generality, let $l(p_2) \leq l(p_1)$. If $l(p_2) < l(p_1)$ there exists a task $p_3 \in N(p_1)$ with $l(p_3) = l(p_2)$. In the case of $l(p_2) = l(p_1)$ we define $p_3 := p_1$.

Now let $S_k = \{p_2, r_2\}$, $S_{k'} = \{p_3, r_3\}$ with $h < k < k' \leq j$. From the choice of the integer h and the properties of S_h follows that the tasks r_2, r_3 must be elements of $N(r)$ and, additionally, $l(p_2) < l(r_2)$ and $l(p_3) < l(r_3)$. Consequently, it results that $l(p_2) < l(r_3)$, since $l(p_3) = l(p_2)$. Hence, task r_3 must be a successor of task r_2, since otherwise, according to the level criterion, r_3 would have been executed instead of p_2 in the time slot k. So we can derive: $l(r_2) > l(r_3) > l(p_2)$. This means that $l(r_2) \geq l(p_2) +2$. But this - as in case 1.) - is a contradiction to the fact that the tasks $r \in V(r_2)$ and $s \in V(p_2)$ are executable in the same time slot h. Hence, this ia a contradiction to the assumption that r does <u>not</u> dominate s. This completes the proof.

<u>Proof of theorem III.1:</u> Let us assume the existance of a non-optimal HU-schedule S over G. Let h be the reducible IOTS with the smallest number in S. Let $p - |S_h| = b > 0$ and $N'(Y_i) := N(Y_i) \cup Y_i - \{e\}$. Then, according to the structure of $\mathcal{G}^*$-graphs,

$U := \{Y_i \in Y | N'(Y_i) \cap S_h = \emptyset\}$ so that $|U| = g > b$.

It follows from Theorem II.1 that

$N(U) \cap \bigcup_{i=h}^{\omega(S)-1} S_i = \emptyset$ and, since h is assumed to be minimal, that $h < \omega(S)-1$.

Hence, there is a task $r \in S_h$ with $l(r) \geq 3$. Let k be the greatest number so that $N(U) \cap S_k \neq \emptyset$, then, k is smaller than h. Let s_o be a task which is an element of $N(U) \cap S_k$. Then, $l(s_o) = 2$. Since $l(r) \geq 3$, there must be a task r' in the set $V(r) \cup S_k$ so that $l(r') \geq 4$. Let $a = t_1 \rightarrow t_2 \rightarrow \dots t_j \dots \rightarrow t_L = e$ be a path with maximum length L in G. Furthermore let

$X_{L-j} := \bigcup_{i \in I_j} S_i$ where the index set I_j is defined as follows:

$I_j := \{i \mid$ there exists a task $t_o \in S_i$ with $l(t_o) = L-j$ and there is no task $t \in S_i$ with $l(t) > l(t_o)\}$ $(0 \leq j \leq L-1)$.

Hence, X_{L-j} is the union of all those S_i that contain a task t_o which has the same level as the corresponding task t_{L-j} on the longest path in G, and that contain no other task with a higher level. Now, from the definition of the sets $X_{L-j} (0 \leq j \leq L-1)$ it follows immediately that for all sets X_{L-j} with $|I_j| \geq 2$ all tasks of the first $|I_j| -1$ time slot, which are elements of X_{L-j} are of level L-j. This means, that all tasks $t \in N'(U)$ which are elements of a set X_{L-j}, are elements of the last S_i in X_{L-j}, i.e. those S_i for which the index i is maximal in L_j. Let S_k be an element of X_{d+3}. Since $r' \in S_k$ and $l(r') \geq 4$, it holds: $d \geq 1$.

From the condition 1.) of this theorem and from $l(s_o) = 2$ it follows: at the beginning of time slot k there exists at least $(d+1)\cdot g$ executable tasks of level 2 to keep busy all processors. We know that $S_h \in X_{d+3}$ with $d \geq 0$, because $r \in S_h$ and $l(r) \geq 3$. Thus, we can conclude that at the beginning of time slot h there exists at least g executable tasks of level 2 to keep busy all processors, too. This contradicts the assumption that h is a reducible IOTS and completes the proof.

1976 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

LIST OF REFEREES

A. Avizienis
R. M. Burstall
J. L. Baer
Kenneth Batcher
P. Bruce Berra
Randal Bryant
Thomas Bredt
Vinton G. Cerf
K. Chandy
I-Hgo Chen
Wei-Tih Cheng
Gregory Chesson
Yaohan Chu
Robert B. Cooper
John A. Cornell
Edward S. Davidson
Edward W. Davis
Jack B. Dennis
Tse-tun Feng
Caxton C. Foster
M. D. Freedman
W. W. Gaertner
Mario J. Gonzalez
William Grosky
C. A. R. Hoare
John Howard
R. C. Holt
J. J. Horning
Carl A. Jensen
Robert W. Johnson
Robert Jump
Dennis Kafura
Leonard Kleinrock
David Kuck
Leslie Lamport
G. Jack Lipovski
Hubert H. Love
Joe M. McKay
W. C. Meilander
R. E. Miller
G. Misunas
D. Scott Parker
C. V. Ramamoorthy
S. S. Reddi
Jerome Rothstein
John S. Sammon
Kenneth J. Schaffer
Howard Jay Siegel
Daniel P. Siewiorek
Hon Hing So
Harold Stone
Kenneth J. Thurber
Ross Towle
Jeffery Ullman
William A. Wulf
David C. Walden
Hisao Yamada
S. S. Yau
Roy J. Zingg

AUTHOR INDEX

Author	Page	Author	Page
Ahuja, S. R.	220	Kain, R. Y.	191
Anderson, G. A.	191	Katz, R.	177
Avizienis, A.	92	Kawamoto, Y.	219
Bandoh, T.	219	Kimsey, D. B.	169
Baqai, I. A.	123	Klayton, A.	189
Batcher, K. E.	65	Krutz, R. L.	221
Boyd, D. L.	83	Krygiel, A. J.	34
Brent, R. P.	254	Lamport, L.	50
Chen, S. C.	196	Lang, R. G.	170
Cheung, L. S.	93	Lang, T.	123
Cotton, J. M.	154, 168	Lawrie, D. H.	283
Daya, M.	246	Leung, W. H.	312
De La Guardia, M. F.	256	Lister, A. M.	43
Erman, R. M.	230	Love, H. H. Jr.	161
Faiss, R.	24	Lyon, J.	24
Feng, T.	30	Maekawa, M.	83
Field, J. A.	256	Majithio, J. C.	303
Francez, N.	235	Mark, M.	168
Friedman, D. P.	263	Meyer, R. A.	18
Gaertner, W. W.	33, 72	Meyer, S. C.	106
Gelly, O.	255	Misunas, D. P.	100, 117
Grosky, W. I.	230	Moellman, D. E.	18
Hand, J.	168	Moshell, M.	222
Hemmersbach, R.	116	Modell, H. S.	247
Hiner, F. P. III	140	Mowle, F. J.	93
Hood, J.	168	Nagel, H. T. Jr.	169
Hsieh, C. P.	304	Nett, E.	317
Jump, J. R.	220	Patel, M. P.	33, 72

AUTHOR INDEX

Author	Page	Author	Page
Pnueli, A.	235	Sparr, T. M.	247
Plas, A.	293	Stanke, E. C.	187
Pearce, R. C.	303	Thomasian, A.	92
Quinn, M.	24	Tjaden, G. S.	55
Reddi, S. S.	33	Towle, R. A.	254
Retter, C. T.	33, 72	Ung, Vincent	213
Reynouard, B.	221	Ward, R. G.	247
Rothstein, J.	206, 222	Weiman, C. F. R.	1
Ruben, S.	24	Wen, K. Y.	283
Sayer, P. J.	43	Wise, D. S.	263
Schaffer, K. L.	145	Yang, C. C.	73
Schutt, D.	116	Vorgrimler, K.	11
Shew, E. S. Y.	154	Zave, P.	35
Siegel, H. J.	273	Zellweger, A.	132
Singh, I. M.	33, 72		