MIT/LCS/TR-454

PARATRAN: A TRANSPARENT, TRANSACTION BASED RUNTIME MECHANISM FOR PARALLEL EXECUTION OF SCHEME

Morry Katz

July 1989

MIT/LCS/TR-454

PARATRAN: A TRANSPARENT, TRANSACTION BASED RUNTIME MECHANISM FOR PARALLEL EXECUTION OF SCHEME

Morry Katz

July 1989

ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme

Morry Katz
(katz@polya.stanford.edu)

Abstract

The number of applications requiring high speed symbolic computation and the performance requirements of these projects are both rapidly increasing. However, the computer science community's ability to produce high performance uniprocessor hardware is being outstripped by these needs. Therefore, we propose a unique multiprocessing solution to the high speed, symbolic computation problem. Our approach is to develop a transparent runtime mechanism for executing standard, sequential Lisp code on a multiprocessor computer. ParaTran, as we call our system, is based on the concept of atomic transactions as developed for use in distributed database systems, programming languages, and operating systems. It utilizes an optimistic scheduling algorithm for processing transactions in order to maximize the available parallelism. In this way, we believe that we can create a system which is both easy to use and yields exceptional performance.

Our concept is based on dividing a Lisp program into a series of pieces, or transactions, which have an a priori sequential order in which they would be executed on a uniprocessor machine. However, instead of performing this serial process, we optimistically run multiple transactions in parallel and then detect at runtime when this parallel execution is not "serializable". An execution ordering is not serializable if it leads to a different result than that which would have arisen from sequential processing. When such conflicts are detected, ParaTran will reexecute certain transactions in order to maintain the edifice of serial computation. Similar approaches using optimistic concurrency have been quite successfully exploited by developers of distributed database and discrete-event simulation systems.

This technical report is a composition of two separate documents: my 1986 masters thesis and a 1987 white paper on extensions to that research. My thesis suggested a number of areas which needed further investigation, many of which were pursued over the following year. The results of continued research led to radical changes being required to the computational model presented in my thesis in order to yield improved parallel performance. My white paper gives an overview of the original computational model, explains the flaws in that model, and then proposes an improved model which yields significantly better performance. My thesis goes into greater detail on the original computational model, as well as discussing several topics that are not addressed in the white paper.

Key Words and Phrases:

Lisp, Scheme, parallel computation, atomic transaction, Time Warp, speculative parallelism, optimistic concurrency.

Table of Contents

Table of Contents

Table of Figures

I. INTRODUCTION

1.1. Overview

ParaTran is a unique runtime mechanism developed by the author for executing standard, sequential Scheme code (a dialect of LISP) [1,2] on a multiprocessor computer. Due to the transparency of the underlying mechanism, the user's view of the system is that of a very high speed uniprocessor engine, a model with which most people utilizing such a system are undoubtedly already comfortable and facile. While hiding the parallelism from the programmer will certainly never lead to the most efficient parallel execution possible, it is the belief of the author that the inherent simplicity of programming in the sequential model and the greater familiarity of users with that model more than offsets the potential loss in efficiency.

The ParaTran mechanism is based on a concept of atomic transactions as investigated in the database literature. It utilizes an 'optimistic' scheduling algorithm in processing these transactions. A program is divided into a series of pieces, or transactions, which have an a priori sequential order in which they would be executed on a uniprocessor. However, instead of performing this serial process, ParaTran optimistically runs multiple transactions in parallel and then detects at runtime when this parallel execution is not 'serializable'. An execution ordering is not serializable if it leads to a different result than that which would have arisen from sequential processing. When such conflicts are detected, ParaTran is forced to reexecute certain transactions in order to effectively serialize the computation.

In the remainder of this chapter, the subset of Scheme which is currently executable by ParaTran and the architectural requirements for the class of multiprocessor on which ParaTran can be implemented will be discussed. Chapter 2 consists of an analysis of alternative approaches to parallel programming. In chapter 3, the complete ParaTran model is presented; and, in chapter 4, the details of building a ParaTran system are considered. Chapter 5 contains a discussion of the results obtained from a simulator of the ParaTran system. Finally, chapter 6 discusses some potential forms of hardware support for ParaTran. A collection of concluding remarks can be found in chapter 7.

1.2. Scheme

1.2.1 Supported Subset

ParaTran supports a subset of Scheme which is a fairly generic representative of the family of Lisp like languages. The data types include integer and rational numbers, booleans, characters, strings, cons cells (pairs), and vectors. The other first class objects in the system are symbols and closures. Continuations are not currently supported, and no consideration has yet been given to the complexity of including them. A strictly lexical scoping scheme is used for finding variable bindings. A standard set of accessors and mutators are available for the different data structures. The furnished language subset includes special forms for performing conditionals, making closures, executing a series of operations sequentially, and creating new environments in the lexical hierarchy.

1.2.2 Programming Style

The ability to use optimistic scheduling of transactions without having to perform a significant amount of backup is based on a perceived programming style within the Lisp community. In the author's experience, programs written in Scheme tend to be largely functional in nature. Side effects are used only when they are absolutely necessary or make a piece of code much easier to write. There is no iteration construct in Scheme; so, all loops are written as recursions. This approach to programming substitutes the creation of new data objects and environments in place of performing side effects on existing ones. It depends greatly on the language's implementation to create objects in the heap efficiently and to reclaim them once they are no longer needed.

1.3. Architectural Requirements

The one characteristic of Scheme which tends to constrain the architectures on which ParaTran could be implemented is its dependence on a single uniform heap. Any object in the heap can have pointers to any other object. This means that some approximation to shared memory is needed if every transaction is going to be capable of running on any processor. True shared memory machines might not be necessary if the heap were partitioned over all of the processors and addresses in the heap were pairs: processor number and local address. In effect, this is just a software implementation of shared memory, and the overhead inherent in such a scheme might actually dwarf the advantage of using a multiprocessor.

II. OTHER APPROACHES TO PARALLELISM

2.1. Functional Programming Languages

Functional programming languages tend to be one of the easier classes of languages to implement. Their functional nature assures that the different computations of which any program is composed can be executed in any arbitrary order, so long as data dependencies are maintained correctly. But, the expressive power of functional languages is somewhat more limited than that of imperative programming languages. There are a number of classes of programs which people desire to write which simply can't be expressed using a completely functional language (e.g. text editors and database managers). Also, there are programs which might be able to be written in a functional style, but are just much more elegantly and easily expressed in an imperative language. (Automated conversion of code from an apparently imperative syntax to a functional semantics could help to ease this latter problem.) For these reasons, the author is interested in a less restricted class of languages.

2.2. Explicit Parallelism Languages

Numerous projects have been undertaken to create languages which contain explicit parallelism constructs. Some of these have involved designing completely new languages incorporating parallelism primitives (e.g. Ada [10]); and, others have been based on the addition of multiprocessing constructs to existing sequential languages (e.g. Multilisp [4,5]). The major problems with explicit parallelism languages all relate to the complexity involved in using them. Since the languages being investigated in such projects are often imperative (i.e. allow side effects), deciding how to decompose a single problem into a number of parallel threads of execution which do not interact in unexpected ways through side effects can be a very difficult task. The complexity of programming in explicit parallelism languages is exacerbated by the fact that one cannot freely use procedural abstraction when coding in them. This is because one has to understand not only the interactions of all of the side effects done by the top level procedure calls in two parallel threads of execution, but also all of those which might be done by any procedures which could be invoked by the top level procedures, etc. Similarly, if one is dealing with a language that has data abstraction, one has to be aware of the implementations of all abstractions that are used in a given program.

A good job of software engineering might tend to alleviate many of the problems stated above. The conditions under which any function could be called could be documented. For example, certain functions might be incapable of being executed concurrently with other functions or other calls to the same function. However, one danger of which the parallel programmer must always be cognizant is that two functions which are equivalent in the sequential domain may not be such in the parallel domain.

Failure to correctly decompose a program into truly independent threads of execution which are guaranteed not to interact unexpectedly through side effects leads to an exceedingly difficult debugging problem. The main cause of this is that the appearance of the bugs tends to be a nondeterministic, and often irreproducible, occurrence. This results from such errors only presenting themselves when the pieces of two independent threads are executed in some particular order. An example of this type of problem would be two processes each of which stored an intermediate result in a common temporary variable and then later retrieved and utilized this result. These processes would only fail to produce their expected outcomes when the execution sequences were interleaved such that each process stored a result in the temporary before either attempted to reclaim its previously stored value. In this case, one of the two threads might retrieve an incorrect value. The instructions in two such processes might only be performed in such a pathological order a small percentage of the time, making this type of bug very hard to locate. Furthermore, its occurrence might likely be dependent on factors out of the user's control, like the scheduling algorithm used by the system in deciding what process to execute on a given processor at any point in time.

A much simpler bug could arise if one process were supposed to store a value in a temporary and then another process were supposed to retrieve the stored value. If these processes were not correctly interlocked and happened to be scheduled in the reverse order, then a value could be read from the temporary before the actual result were ever stored. This could cause the second process to use an erroneous value in its computation. In fact, due to idiosyncrasies in the interaction between the scheduling algorithm used by a given system and a piece of code which spawned two such processes, a program of the type just described might actually work correctly on a machine with N processors but fail on one with N+1 processors. This means this type of bug could lie dormant in a crucial piece of code for years and then suddenly appear when a program was executed on a new or upgraded machine.

Maintenance of programs written in explicit parallelism programming languages is also quite complex. This is due to the need to understand how any single update to a program might affect every other portion of the code. Great care must be taken to assure that the insidious types of errors described so far are not introduced into an otherwise correctly functioning program. The author believes that for real world systems this would often require understanding an inordinately large fraction of a program, even to make just a small, apparently localized, modification. While it has been admitted that good documentation and appropriate programming style might significantly lessen many of the problems with explicit parallelism languages, it is the further belief of the author that they cannot be eliminated completely and will be a perpetual source of annoyance, requiring much time and effort to detect and correct.

2.3. Compile Time Detection of Parallelism

A yearning to maintain the sequential programming model in conjunction with the "dusty deck" problem has motivated several groups to work on compile time detection of parallelism in code written in sequential languages (e.g. [7]). The major shortcoming of this approach, in the view of the author, is that there is much parallelism that exists in a program which cannot be extracted at compile time. If there is any possibility, no matter how remote, that two pieces of code might yield an incorrect result if run in parallel, then a compiler designed to generate parallel code must make a conservative decision and ensure that two such tasks are executed sequentially. In the author's experience, it is often the case that the great majority of the time such code could actually have been run in parallel. This loss of potential parallelism by compiler based systems can lead to trouble generating code to utilize a machine with a very large number of processing units. A more sophisticated compiler might detect cases of high probability parallelism and generate the appropriate runtime checks to allow conditional parallelization; however, a successful system of this type has yet to be developed.

Much of the work done in compile time detection of parallelism is done in languages like Fortran which have a fairly simple programming model. Compile time analysis is much less effective for languages in the Lisp family because their increased expressive power is gained at the expense of a more complex programming model. Scheme programs tend not to have the simple structure exhibited by many examples of Fortran code. They typically aren't built up of simple loops with fixed bounds, and their flow of control is often very data-dependent. Also, in Scheme, code can be generated at runtime, and the bindings between function names and code are delayed until execution time. This last problem can be partially overcome by compiler directives to do early bindings of some function names and bodies; but, it is the overall conclusion of the author that compile time analysis is not sufficient to generate highly parallel Scheme code.

2.4. ParaTran

The advantages of the ParaTran approach over the alternatives are that the programming model is sequential, side effects are allowed in the semantics of the language, and the parallelism is detected at runtime when more efficient decisions can be made about what can be run in parallel. The major disadvantage of ParaTran is the potential waste caused by its speculative nature. Increased parallelism is achieved by running pieces of code concurrently before it can be ascertained whether this will yield the desired sequential result. When a violation of the serial model is detected, the results of certain processes must be thrown out and their associated code reexecuted. This can lead to conceivably costly overhead if there are numerous interactions between processes.

III. THE PARATRAN MODEL

3.1. Introduction

The ParaTran system will be described by analogy with atomic transactions as used in database systems. First a brief discussion of atomic transactions and an optimistic scheduling method for their implementation will be presented. This will be followed by an explanation of how the execution of sequential programs can be viewed as a series of transactions with a database system. Once this groundwork has been laid, an examination of how to decompose a program into transactions will be undertaken. This chapter will conclude with an illumination of some of the actual details of the ParaTran model of computation.

3.2. Atomic Transactions

An **atomic transaction** as used in the context of database systems is a set of operations to be performed on a database in such a manner that the entire transaction appears to take place as a single operation. In other words, an atomic transaction takes a database from one consistent state to another. The only two states of the database which are visible to other transactions are those which existed before the given transaction was initiated and after it was completed. An example of the utility of atomic transactions would be an attempt to transfer money between two accounts in a bank database. Such a transaction is composed of two operations: the removal of the funds from the first account and their deposit into the second. The database should never be visible in a state in which the transferred funds are in neither account, or in both. Therefore, the transfer must be atomic.

An 'optimistic' scheduling mechanism for implementing atomic transactions (to be referred to from now on merely as transactions) can be built using a three phase protocol [8]. This protocol depends on the maintenance of two data structures by each transaction: a **read list** and a **write list**. The read list is a record of all database entries read by a transaction; and, the write list, a record of all database updates to be performed by the same.

The first phase of the three phase protocol is the **read** phase. During the read phase, a test run of the entire transaction is performed. All database updates attempted during the read phase are recorded in local copies of the database entries to be modified and kept in a write list. All database queries during the read phase are directed to the actual database unless a local copy of the entry to be read exists due to a prior update by this transaction. In this case, the local value is utilized. A record of all entries in the actual database which are referenced during the read phase is kept in a transaction's read list.

The second phase is **validation** and consists of checking that all of the database queries performed during the read phase returned values which are consistent with the current state of the database. If it is found that the database still contains the same values which were read earlier, then the transaction succeeds in validating. Otherwise, the transaction **aborts** and is begun anew by repeating the read phase. Finally, during the **write** phase the actual database is updated based on the local copies of the database entries to be modified in a transaction's write list. In order to ensure correct operation of this mechanism, the validate and write phases of every transaction must be done as a single atomic operation. Furthermore, only a single transaction can be validating or writing at any instant.

Once one has implemented a transaction based system, it is desirable to be able to build up what are called **nested transactions** [9,11]. These are a group of transactions which are to be combined into a single atomic unit. Not only are the subtransactions of this new transaction supposed to execute atomically with respect to each other; but, the entire group should appear to happen atomically as viewed by other transactions. Given the ability to nest transactions one level deep, there is no reason not to generalize this to arbitrary nesting of transactions. In figure 3.1, a tree is used to represent a single transaction formed from three subtransactions. The first and third subtransactions are themselves formed of two subsubtransactions. In order to maintain a simple semantics for nested transactions, it will be assumed that if a transaction has subtransactions, all of the computation is done in the subtransactions. The supertransaction's only purpose is to join its subtransactions into a single atomic unit. (i.e. All computation is done in the leaves of the transaction tree.) This restriction can and will be relaxed slightly in some later descriptions.

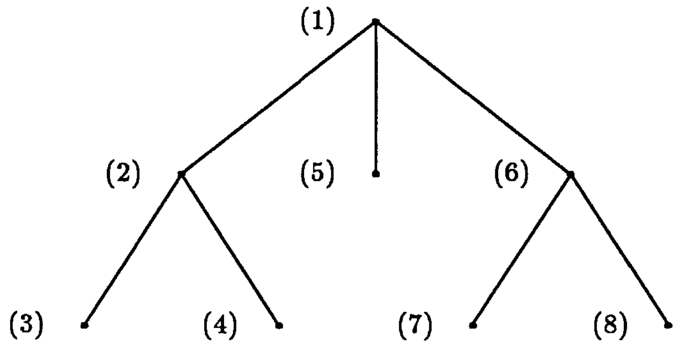


Figure 3.1. A tree representation of nested transactions

The implementation of nested transactions is a fairly logical extension of the one for simple transaction systems. The read phase of the leaf transactions is performed in an identical manner to that previously presented. The first subtransaction of any supertransaction to attempt to validate succeeds automatically. During its write phase, rather than making the updates specified in its write list to the actual data base, it sends its local copies of database entries to be modified to its parent transaction where they become the parent's write list. Also, this child's read list becomes its parent's read list. Sibling transactions of the first subtransaction to validate are validated by checking that they have not done any queries to records which were updated by siblings which have already completed their validate and write phases. This is determined by comparing their read list with their parent's write list. If a match is found then the given subtransaction fails to validate and must be restarted with its read phase. Before it is restarted, the aborted subtransaction's read list must be cleared and write list must be initialized to contain the current contents of its parent's write list. If this passing down of updates were not done, then a subtransaction which read an entry updated by a sibling which validated first would continue aborting and restarting ad infinitum.

On the other hand, if a subtransaction succeeds in validating then its read and write lists are merged into those of its parent during its write phase. Once all of the subtransactions of a single transactions have **completed** (i.e. finished validating and writing), the parent transaction is left with a list of queries which were made by its children, and all of the updates done by the same. This state has the same form as that of any of the original leaf transactions, so the parent transaction can now be validated relative to its siblings in the same manner as was described above. The top level transaction is treated as it would have been in the original transaction model in that its writes are actually made to the physical database.

The restriction that only one transaction can be completed at once can now be weakened somewhat so that multiple descendants of a single transaction can be completing in parallel so long as none of these are siblings. From an analysis of this scheme, it is evident that if a transaction fails to validate, any work done by its descendant transactions is lost, even if their execution was unrelated to the cause of the abort. Therefore, there can be a fairly heavy performance penalty when deeply nested transactions are aborted.

As an example of the use of nested transactions, assume that in figure 3.1 the following operations are performed:

Transaction 3 reads entry A and writes entry B.

Transaction 4 reads entry B.

Transaction 5 writes entry C and then reads it.

Transaction 7 reads entry D.

Transaction 8 reads entry B.

For simplicity, it will be assumed that the transactions in figure 3.1 validate sequentially and validation of subtransactions will be attempted from left to right. Based on these assumptions, the first transaction to attempt to validate will be transaction 3; and, since no transactions have already completed, transaction 3 will succeed in validating. It passes its read list containing A and its write list containing B to it parent, transaction 2. Transaction 4 will now fail to validate because its read list contains B which is now on the write list of transaction 2. Consequently, transaction 4 must be aborted. It will be restarted with the write list of its parent, transaction 2, so that its read of B will access the local copy rather that the global one. Also, B will not be placed on the read list of transaction 4 this time since B is already in its write list. Transaction 4 will now validate, and during its write phase will add its read and write lists to transaction 2's. Since transaction 4's lists have no entries not already included in the lists of transaction 2, this operation has no net effect. Transaction 2 is now ready to validate and will succeed in doing such since no other subtransactions of transaction 1 have already validated. During its write phase, transaction 2 will pass its reads and writes to transaction 1. Therefore, the read list of transaction 1 will contain A; and the write list, B. The write list for transaction 5 contains C; but, its read list is empty since it read C after it wrote it. This means transaction 5 will validate and add its write list to transaction 1's. The write list of transaction 1 will now contain both B and C. Transaction 7 will validate because no other child of transaction 6 has already completed. It will pass a read list containing D and an empty write list to transaction 6. Since the read list of transaction 6 is empty, transaction 8 will validate, and its write phase will cause the read list of transaction 6 to contain both B and D. Transaction 6 will now fail to validate since the write list of transaction 1 contains B. This will cause transaction 6 to abort and be restarted with the write list of transaction 1. It will in turn pass this write list to transactions 7 and 8 when they are restarted. Once transactions 7 and 8 have been completed again, transaction 6 will only have D on its read list. It will validate causing the read list of transaction 1 to contain A and D, and the write list to contain B. If no other top level transactions have been executing, then A and D couldn't have been modified; therefore, transaction 1 will succeed in validating, and during its write phase will update entry B in the global database. Otherwise, transaction 1 might fail to validate, requiring the entire process to be reinitiated.

It should be noted that in the above example transaction 6 was aborted and transactions 7 and 8 had to be run a second time all because of the read of B performed by transaction 8. This should serve as a demonstration of the potential inefficiency of this form of nested transactions. Ideally, only transaction 8 should need to be aborted and rerun.

The nested transaction model presented thus far has had no required ordering amongst sibling transactions. Effectively sequential processing of subtransactions is often desirable. It can be implemented by serializing the validation order of sibling transactions as was done in the above example. Each transaction is required to wait for the sibling on its left to validate before it validates. An apparently sequential left to right execution of all of the leaf transactions in a transaction tree results.

3.3. Transaction Analogy

An entire program can be viewed as a set of transactions, if pieces of code, or tasks in the traditional programming language terminology, are viewed as transactions against a database which is the global store. If the bodies of functions are subdivided into a number of transactions which themselves call other functions, then a nested transaction model results. In the description of transactions against database systems previously presented, ordering constraints between sibling transactions were optional. However, in the sequential programming language domain there is an a priori ordering of events which must be obeyed. If a complete ordering is enforced on all of the transactions within a transaction tree, then the tree really reduces to a list of transactions. These transactions can be performed as though there were no nesting. Such a sequentialization of transactions is performed on the transaction trees of programs executed by ParaTran so that the inefficiency of losing all subtransactions whenever a supertransaction is aborted can be removed. It should be noted, however, that this approach can lead to other problems which are discussed later.

A tree structure for storing transactions is preserved because it is a convenient means of retaining information about which task spawned any given task, and because it has nice localized locking properties. Since a task which performs a function call can itself spawn a number of subtasks, it is necessary to be able to splice tasks into the middle of the conceptually serial task list. If multiple processors are to be able to do such modifications to the task structure in parallel, then some form of localized locking is necessary. It happens that trees are an ideal structure to allow insertion and deletion in parallel.

The execution of tasks under ParaTran is divided into the same three phases as for transactions. During the read phase, the code for a task is executed. All interrogations of variable bindings and data structures are recorded as queries to the database. Attempts to modify variable bindings or data structures are side effects which are viewed as database writes and are shadowed in the same manner as they would be for a transaction system. Validation is now performed in the sequential order in which the transactions are supposed to appear to be executing. The first transaction must succeed in validating since it starts with a global store which couldn't have been modified by any other task before this first task has completed. During its write phase, the first task updates the global store based on its shadowed side effects and then broadcasts its list of modified objects to all other tasks in the system. These tasks do a partial validation by comparing their read lists with the writes broadcast by their predecessor. If a match is found, the given task will fail to validate, so it is immediately aborted and its execution is restarted from the beginning. Once a task becomes the next one to validate and has processed all of the broadcast messages it has received, it has already aborted if necessary. Therefore, this task must automatically succeed in validating once it has finished executing.

3.4. Segmentation of Programs into Tasks

An initial segmentation methodology will be presented which leads to a very fine grain parallelism. Such a decomposition of programs would undoubtedly add an inordinate amount of overhead to the time required for execution. Therefore, as this description progresses, modifications to the basic model will be described which are conducive to reducing this overhead.

Basically, Scheme programs are created utilizing only four constructs: function creation, function invocation, sequentialization, and conditionals. The sequentialization construct, **sequence**, is composed of a series of subexpressions. Due to the functional nature of the language, a sequence returns the value of its last subexpression. In the initial segmentation model, a transaction is built for the entire sequence, and a subtransaction is created for each subexpression of the sequence. A transaction tree for the sequence (**sequence a b c d**) can be found in figure 3.2. The root node of the tree represents the transaction for evaluating the entire sequence, and each of the leaves is a subtransaction for evaluating one of the subexpressions. The order in which the subexpressions which compose a sequence would have been executed on a uniprocessor can be reconstructed by reading the children in the transaction tree from left to right.

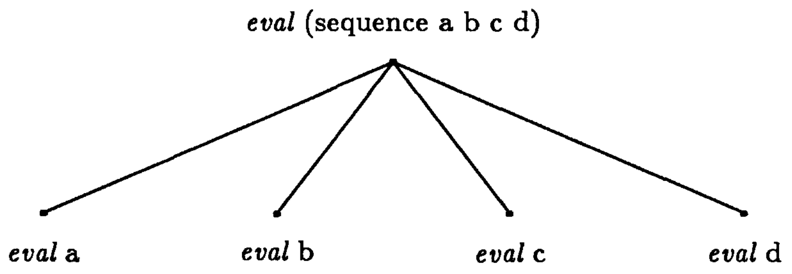


Figure 3.2. A transaction tree for (sequence a b c d)

A better understanding of ParaTran transaction trees is achieved by analyzing Scheme code which could generate them. In figure 3.3 the code for producing the transaction tree in figure 3.2 is presented. (See appendix 1 for a short description of Scheme.) **Spawn-task** is a macro which expands into a call to a procedure, **spawn**, which spawns a new subtask of its parent and makes it the rightmost child of its parent. **Spawn** takes a single argument which is a thunk for evaluating the spawned task. The code for the thunk is a sequence generated from the arguments to **spawn-task**. The call to **spawn** returns a placeholder into which the value returned by the thunk will be placed. The mechanism for returning values from the task spawns and for synchronizing the creation and use of these values will be discussed in the next section. (See figure 3.4 for a macro for **spawn-task** in MIT Scheme syntax and an example of its use.)

```
(spawn-task
 (spawn-task a)
 (spawn-task b)
 (spawn-task c)
 (spawn-task d)))
```

Figure 3.3. Code for producing a transaction tree for (sequence a b c d)

In the case of a sequence, no actual computation is done in the root transaction. All of the work is done in the subtransactions, and these must be validated and writ-

```
(define-macro (spawn-task .  args)
  '(spawn (lambda () ,@args)))

(define-macro (spawn-application-task .  args)
  '(spawn-application (lambda () ,@args)))

(spawn-task (+ 1 2) 'end)
   expands to
(spawn (lambda () (+ 1 2) 'end))
```

Figure 3.4. Macros for spawning tasks and an example of their use

ten in the specified left to right order. Since the subtransactions for the individual subexpressions might themselves be composed of multiple subtransactions, doing the validate and write phases for a subtransaction might involve actually doing validation and writing for numerous descendants of the given transaction. Once a transaction has **completed**, (i.e. all three phases of the transaction have been performed), the transaction is removed from the transaction tree in order to free the memory used to store this portion of the tree. The transaction for the entire sequence does not need to be validated or written since transactions are not conceptually nested under ParaTran. However, the node in the transaction tree for the sequence should be removed once the nodes for all of its children have been removed.

The invocation of a function can be broken into three parts. First, the operator must be evaluated to determine what code should be used in the function application. This is necessary in the Lisp family of languages because the code for a function may actually be the result of executing some other piece of code. Next, each of the operands must be evaluated. These too could be the results of other computations. Once both the operator and operands are known, the actual function application can be performed. Consequently, a function call becomes a transaction for the evaluation of the function call which is composed of a series of subtransactions: one for the evaluation of the operator, one for the evaluation of each operand, and one for the true function application. Although Scheme does not impose any ordering constraints on the evaluation of the operator and operands, ParaTran enforces an effective evaluation ordering amongst these forms. This is done to simplify debugging by limiting nondeterminism. Given some total ordering for the evaluation of the operator and operands of each function call, most programs will execute in the same manner and give the same result if given the same input. By effectively sequentializing evaluation, ParaTran makes debugging on a multiprocessor more deterministic. For simplicity, the operator will be evaluated first followed by a left to right evaluation of the operands in all examples in this thesis. A ParaTran compiler might actually select different orders of evaluation for some function calls in order to improve efficiency.

An example of a transaction tree created for a simple function call of the form **(append a b)** can be found in figure 3.5; and, code for generating the tree, in figure 3.6. The root of such a tree represents the parent transaction and each of its children represents a single subtransaction. All of the actual computation in such a tree is done at the leaf nodes, and the sequential order in which the tasks would have executed on a uniprocessor can again be reconstructed by reading the children from left to right. This means the child tasks should be completed from left to right, and each should be removed from the transaction tree as soon as it completes. As with sequences, the parent task should be removed from the tree as soon as all of its children have been removed.

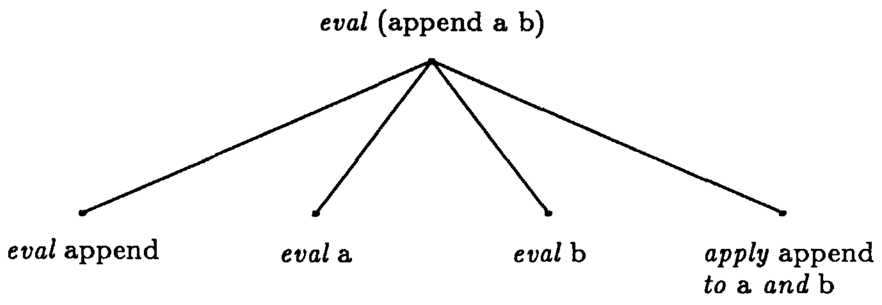


Figure 3.5. A transaction tree for **(append a b)**

There is a very important piece of work which is done as part of a function application and adds to the complexity of the ParaTran model. This is that the function to be applied must be read before the function application can be performed. This read is one which must be recorded during the read phase of the application transaction. (The mechanisms for read and write logging will be discussed in detail in the next chapter.)

```
(spawn-task
 (let* ((*operator* (spawn-task append))
        (*operand1* (spawn-task a))
        (*operand2* (spawn-task b)))
   (spawn-application-task
    (*operator* *operand1* *operand2*))))
```

Figure 3.6. Code for producing a transaction tree for (append a b)

Clearly, if an incorrect binding between a function name and body is utilized in executing a function application, the transaction for that application must be aborted. The transaction should then be restarted using the correct function code. (A sample piece of code which might necessitate such an abort can be found in figure 3.7.)

```
(sequence
  (set! append +)
  (set! append cons)
  (append a b))
```

Figure 3.7. Sample code which might cause an abort

In the event that the transaction for the function application is itself composed of multiple subtransactions, these subtransactions may no longer be meaningful once the parent has been aborted and should be **killed**. When a transaction is killed, it is permanently removed from the transaction tree and the system, instead of merely being restarted as would be the case for an abort. This is the reason that information on the task hierarchy had to be maintained even once the task tree had been converted to an object much more resembling a list. The subtle point in all of the above is that the correctness of the function binding used must be validated prior to validating any of the children of the actual application subtransaction. This ensures that no child of an application subtransaction is ever completed when in reality it should have been killed as a result of its parent aborting due to the use of an incorrect function binding. The reading of the function binding is always done in the application subtransaction, not in a subtransaction of that transaction. This differs from the basic nested transaction model in which all of the reading, writing, and computation for a single transaction is done in its subtransactions, if it has any. The application subtransaction must now serve two purposes. It acts as a parent to its children, and the computation done in the application subtransaction acts as its own first child. Validation of the function binding is achieved by just validating application subtransactions before their children. Since it can't be determined syntactically whether an application transaction will spawn subtransactions, it is advantageous to have application transactions always perform their write phase after they validate. If the application transaction has subtransactions then nothing will happen during the write phase; otherwise, the validation and write phases will effectively be performed on this node just like any other leaf node.

While the handling of function applications seems to be very complicated, it yields great dividends in that conditionals can be implemented using the identical mechanism. Evaluation of a conditional involves two steps: evaluating the predicate and evaluating either the consequent or the alternative. ParaTran creates a transaction for every conditional and subtransactions for each of the two steps in evaluating the conditional. A transaction tree for a conditional of the form **(if pred a b)** can be found in figure 3.8 and code for generating the tree in figure 3.9. The subtransaction for the evaluation of the consequent or alternative can be handled just like an application transaction. This subtransaction has the same type of dependency on the predicate that an application has on its operator. If the value returned by a predicate changes due to the predicate transaction being aborted, and reexecuted, then the transaction for the alternative or the consequent needs to be aborted and any children it might have spawned must be killed. Similarly, the value returned by the predicate must be validated before any children of the consequent or alternative transactions can safely be completed.

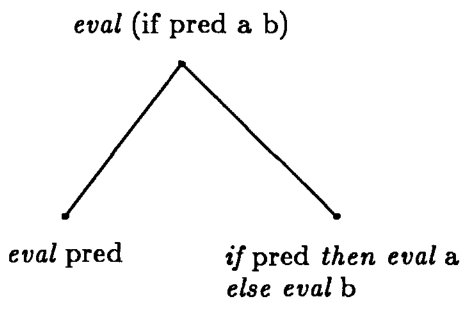


Figure 3.8. A transaction tree for **(if pred a b)**

```
(spawn-task
 (let* ((*pred* (spawn-task pred))
   (spawn-application-task
    (if *pred* a b))))
```

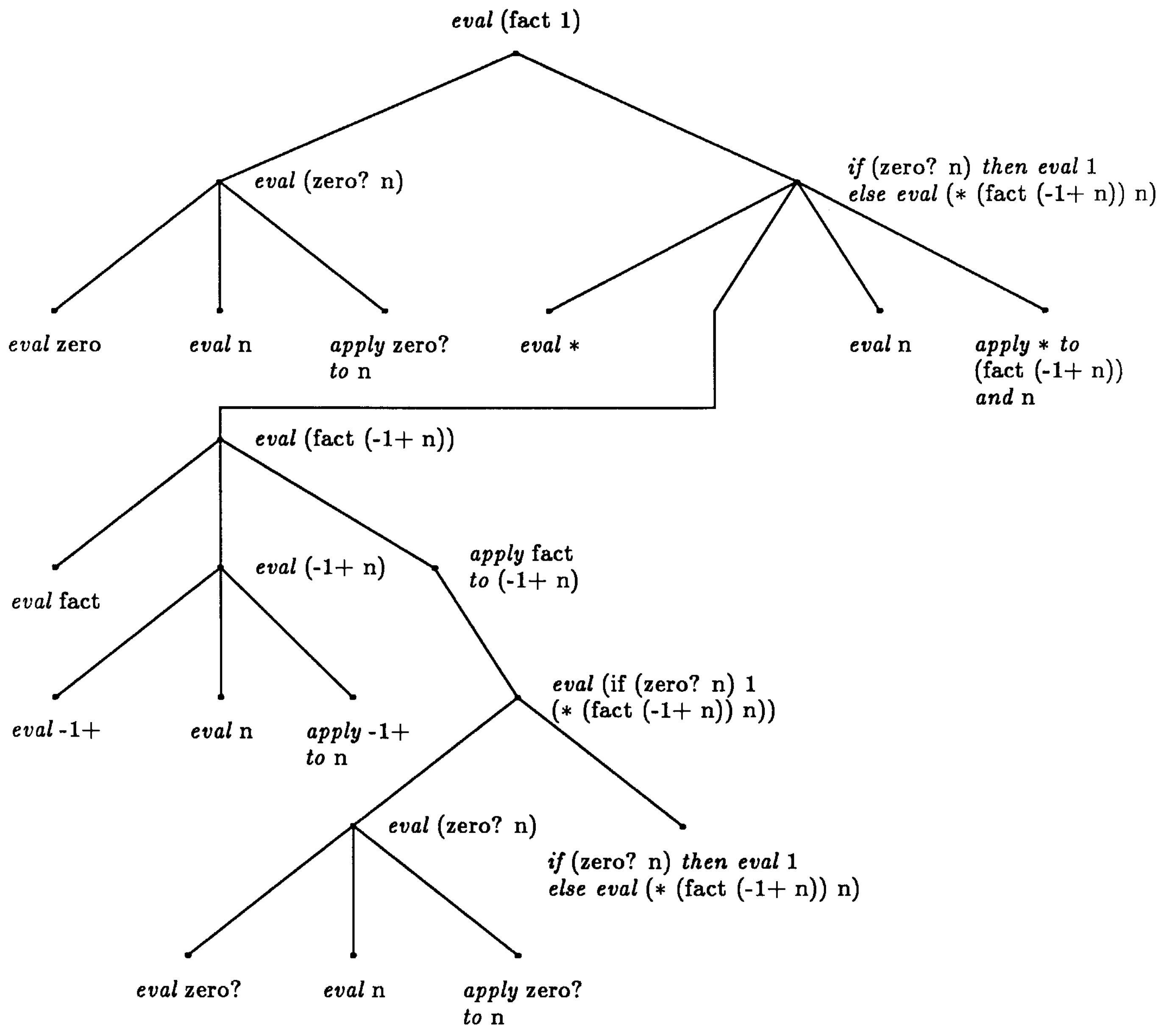
Figure 3.9. Code for producing a transaction tree for (if pred a b)

```
(define (fact n)
  (if (zero?  n)
      1
      (* (fact (-1+ n)) n)))
```

Figure 3.10. A recursive factorial program

```
(define (fact n)
  (let* ((*pred* (spawn-task
                  (let* ((*operator* (spawn-task zero?))
                         (*operand* (spawn-task n)))
                    (spawn-application-task
                     (*operator* *operand*))))))
    (spawn-application-task
     (if *pred*
       1
       (spawn-task
        (let* ((*operator* (spawn-task *))
               (*operand1*
                (spawn-task
                 (let* ((*operator* (spawn-task fact))
                        (*operand*
                         (spawn-task
                          (let* ((*operator* (spawn-task -1+))
                                 (*operand* (spawn-task n)))
                            (spawn-application-task
                             (*operator* *operand*))))))
                   (spawn-application-task
                    (*operator* *operand*)))))
               (*operand2* (spawn-task (lambda () n))))
          (spawn-application-task
           (*operator* *operand1* *operand2*)))))))
```

Figure 3.11. Code for producing a transaction tree for (fact n)



eval (fact 1)
eval (zero? n)
if (zero? n) *then eval* 1
else eval (* (fact (-1+ n)) n)
eval zero
eval n
apply zero? *to* n
eval *
eval n
apply * *to* (fact (-1+ n)) *and* n
eval (fact (-1+ n))
eval fact
eval (-1+ n)
apply fact *to* (-1+ n)
eval -1+
eval n
apply -1+ *to* n
eval (if (zero? n) 1 (* (fact (-1+ n)) n))
eval (zero? n)
if (zero? n) *then eval* 1
else eval (* (fact (-1+ n)) n)
eval zero?
eval n
apply zero? *to* n

Figure 3.12. A transaction tree for **(fact 1)**

A simple recursive factorial program is presented in figure 3.10. The transaction tree created for the function invocation **(fact 1)** can be found in figure 3.12, and the code for generating one level of the tree can be found in figure 3.11.

3.5. Futures

In keeping with the functional style of Scheme, every task into which a program is divided must return a value. Since many operations in a functional language only manipulate references to an object and never actually look at the object itself, some parallelism can be gained between function application and argument evaluation if tasks return placeholders into which they will store their results once they are calculated. **Futures** are a type of placeholder which have a special set of properties that make them desirable for this application [4,5]. A future is initially created without any value assigned to it and is said to be **undetermined.** If a task tries to refer to the value of an undetermined future, it becomes suspended and is placed on a queue of tasks waiting for a value to be assigned to the future. When a value is assigned to a future, the future becomes **determined.** All of the tasks which were waiting on the calculation of the future's value are now returned to a general work queue of tasks to be executed. The execution of these tasks is resumed from the point at which they attempted to access the value of the just determined future. An attempt to access the value of a determined future is completely transparent. The value is returned just as though the future object weren't present.

A presentation in box-and-pointer notation [1] of the Scheme objects used to represent undetermined and determined futures can be found in figures 3.13 and 3.14. For each rectangle, the left hand side contains the type of the object and the right hand side, the value. If the value is a pointer then the actual value is a set of consecutive Scheme words beginning at the location to which the value points. (See appendix 2 for more details on the conventional manner in which objects are stored in the heap by Scheme.)

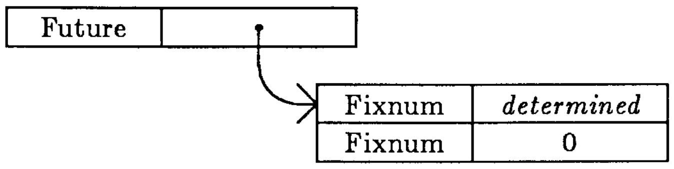


Figure 3.13. The Scheme representation for a determined future

Once a future becomes determined, the future object is no longer necessary. In order to save memory and increase execution speed, determined futures are **spliced out**

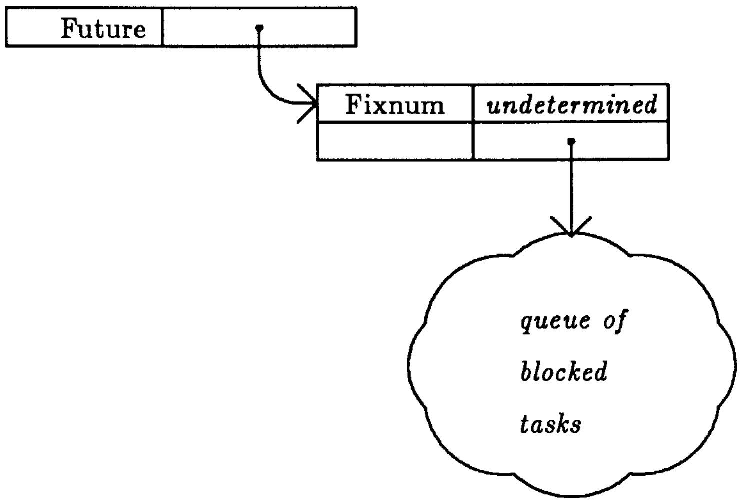


Figure 3.14. The Scheme representation for an undetermined future

by the garbage collector. Splicing out a future means that all pointers to the future are replaced by pointers to the value assigned to the future. This form of splicing maintains all sharing properties inherent in the semantics of the original language. In figure 3.15, the garbage collector splices out a future from a cons cell (pair) whose car (first component) is the number 0 and whose cdr (second component) is a determined future whose value is the number 1.

If futures are to be used as the interlocking mechanism between different tasks under the ParaTran model, a decision must be made as to when futures actually become determined. Although a result value is computed for a task as soon as its execution, or read phase, finishes, this value is not actually guaranteed to be correct until the task completes, through the culmination of its validation and write phases. This means that the result value of a task must not be stored in its future and the future made determined until the write phase. Otherwise, a future which contained an incorrect value could be spliced out by the garbage collector before a correct value could be substituted for the erroneous one originally placed in the future.

The need to do a late instantiation of values to futures conflicts with the desire to maintain maximal parallelism. This is because the evaluation of the operator to be used in a function application is done in a separate task from the actual application (refer back to Figure 3.5). If the value of an operator (i.e. its code) cannot be returned

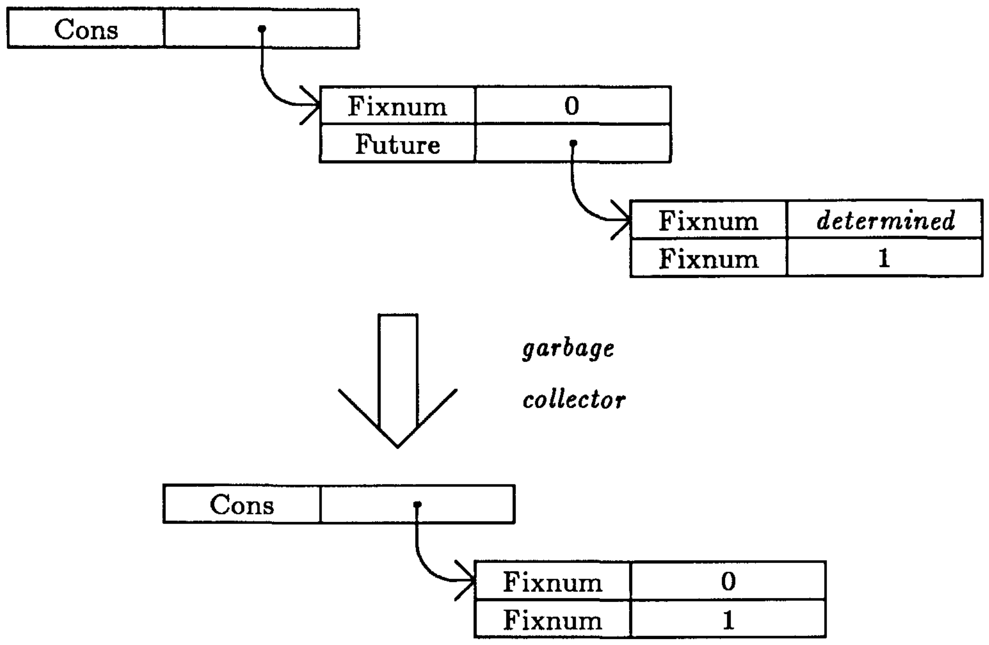


Figure 3.15. The splicing out of a determined future by the garbage collector

in a future until the task to evaluate the operator completes, then almost no overlap of function evaluation and function application can take place. This means that the only parallelism remaining in a function evaluation is parallel argument evaluation. However, if the evaluation of the arguments themselves are composed of other function applications, they too will be sequentialized due to the need for each operator task to be completed before the associated application task can be executed. (Remember that the completion order is sequential.) A similar problem exists for conditionals. Since separate tasks evaluate the predicate of a conditional and either the alternative or consequent (refer back to Figure 3.8), the evaluation of either branch of a conditional could never be initiated until the predicate were completed. (This assumes that the consequent and the alternative are not both spawned aggressively; but, either one or the other is spawned based on the predicate's value.) As a result, every conditional would act as a sequentialization point. The above two restrictions tend to remove nearly all the parallelism from typical programs and are therefore unacceptable.

A solution to this problem is the utilization of **keep slot** futures. These are futures which have an intermediate state between being undetermined and being determined.

When a future becomes a keep slot, it has a value, but it cannot be spliced out by the garbage collector. If the value of a keep slot future is requested, it is returned just as it would have been for a determined future. However, since a keep slot cannot be spliced out by the garbage collector, the future object always remains; and therefore, its value can be changed. Keep slot futures are the actual constructs which are used for interlocking tasks in ParaTran. When a task is spawned, it creates an undetermined future into which its eventual result will be stored. When the task finishes executing for the first time, its converts the future from being undetermined to being a keep slot and stores the computed return value in the future. Finally, during a task's write phase, when its return value is assured to be correct, the task's future is converted to the determined state.

In order for ParaTran to work correctly with keep slot futures, some logging of reads of futures and writes to futures must be done. Each time a read of the value of a keep slot future is performed, a read of the future object must be logged. This helps assure that if the value of a keep slot future is changed, all of the tasks which have read the old value will be aborted. If a task is aborted after it has finished running the first time and consequently converted its future to a keep slot, then each subsequent time the given task finishes executing, it just assigns a new value to the future and logs the fact that it has written to the future object. It is this write which will cause any readers of a changed keep slot to be aborted.

The logging of writes to keep slot futures is only really necessary if the value returned when a task is reexecuted differs from the value returned the previous time. Since predicates of conditionals only return one of two values, true or false, there is a fairly high probability that a second execution of a predicate will return the same value as the first, regardless of what piece of code is run for the predicate. Based on this observation, the optimization of only logging writes when the value returned actually changes could often save unnecessary aborting of the consequent or alternative of a conditional when the predicate aborts. The one problem that persists is that any object in Scheme which is not the false object is considered to be true. Consequently, not all true objects are the same. In order to insure a minimal level of aborts, one might wrap all predicates in a function which converts their results to unique true and false objects.

3.6. Sample Run of ParaTran

In order to fully understand the operation of the ParaTran runtime mechanism, one must go through at least one example step by step. A fairly simple sample program can be found in figure 3.16. The code for generating a transaction tree for the sample program in figure 3.16 can be found in figure 3.17. Finally, a possible transaction tree generated by this code can be found in figure 3.18.

```
(sequence
  (set! flag #!true)
  (set! flag #!false)
  (if flag
      (cons flag 'incorrect-operation)
      (cons flag 'correct-operation)))
```

Figure 3.16. A sample program using side effects

```
(spawn-task
 (spawn-task (set! flag #!true))
 (spawn-task (set! flag #!false))
 (spawn-task
  (let* ((*pred* (spawn-task flag)))
    (spawn-application-task
     (if *pred*
         (spawn-task
          (let* ((*operator* (spawn-task cons))
                 (*operand1* (spawn-task flag)))
            (spawn-application-task
             (*operator* *operand1* 'incorrect-operation))))
         (spawn-task
          (let* ((*operator* (spawn-task cons))
                (*operand1* (spawn-task flag)))
            (spawn-application-task
             (*operator* *operand1* 'correct-operation))))))))
```

Figure 3.17. Code for producing a transaction tree for the side effect program

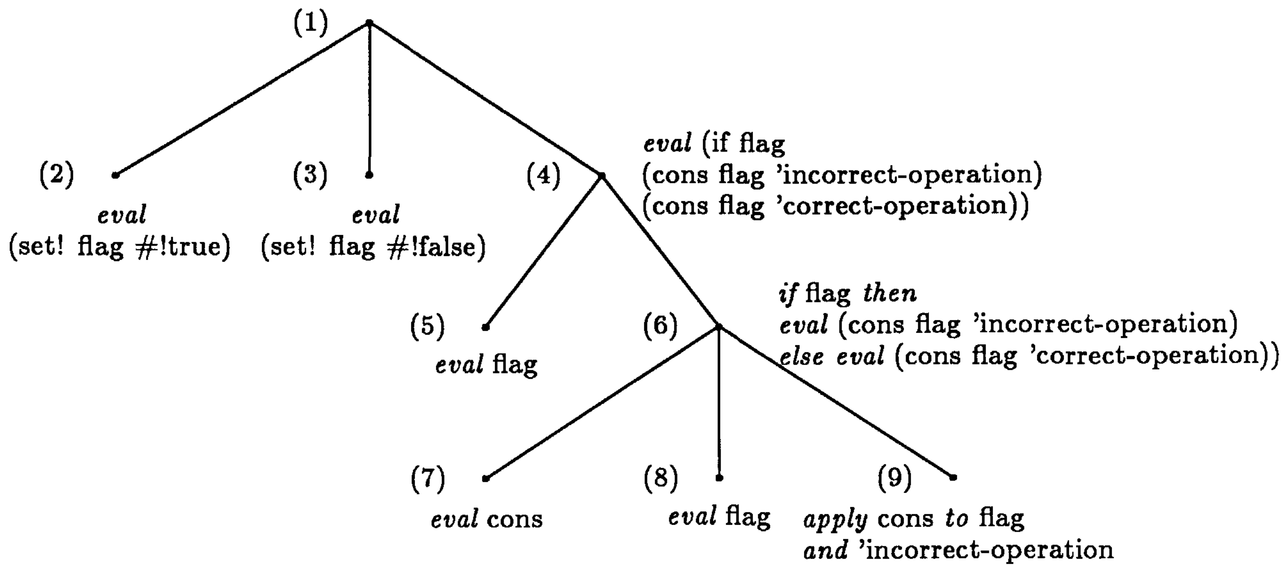

(1)
(2)
eval
(set! flag #!true)
(3)
eval
(set! flag #!false)
(4)
eval (if flag
(cons flag 'incorrect-operation)
(cons flag 'correct-operation))
(5)
eval flag
(6)
if flag *then*
eval (cons flag 'incorrect-operation)
else eval (cons flag 'correct-operation))
(7)
eval cons
(8)
eval flag
(9)
apply cons *to* flag
and 'incorrect-operation

Figure 3.18. A transaction tree for the side effect program

The transaction tree found in figure 3.18 could have been generated by the following execution sequence. Task 1 spawns tasks 2, 3, and 4. Task 2 performs its read, validate, and write phases. Before task 2 is removed from the transaction tree, task 4 spawns tasks 5 and 6, and task 5 performs its read phase and returns the current binding of flag, which is true, in its future. This enables task 6, which might have been waiting on task 5's future, to continue its read phase and spawn tasks 7, 8, and 9. At this point the transaction tree in figure 3.18 has been built.

All of the tasks are now free to perform their read phases. However, if tasks are to be completed, this process must continue from the point at which task 2 was about to be removed from the transaction tree. Following this operation, task 3 could be completed. Its write phase would cause a broadcast of the changing of the binding of flag. Once task 3 has been completed and removed from the transaction tree, task 5 attempts to validate and is aborted due to the broadcast by task 3. When task 5 has finished its read phase for the second time, task 5 can be completed. This time task 5 has found the binding of flag to be true. Since this is a different value than was returned by task 5 the first time, completing task 5 will cause a broadcast of the modification of task 5's future object. This broadcast causes task 6, which has read task 5's return value, to be aborted. Since task 6 is an application task, it will have attempted to validate, and therefore be aborted, before tasks 7, 8, or 9 try to complete. Aborting task 6 causes tasks 7, 8, and 9 to be killed and removed from the task tree. Task 6 now executes for a second time and spawns tasks 10, 11, and 12. The transaction tree which results is shown in figure 3.19.

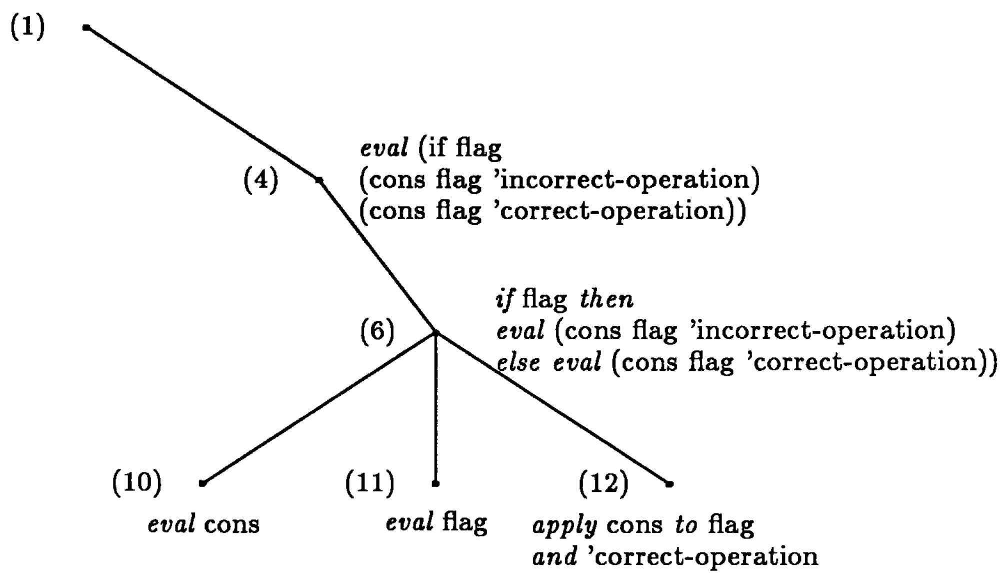

(1)
(4)
eval (if flag
(cons flag 'incorrect-operation)
(cons flag 'correct-operation))
(6)
if flag *then*
eval (cons flag 'incorrect-operation)
else eval (cons flag 'correct-operation))
(10)
eval cons
(11)
eval flag
(12)
apply cons *to* flag
and 'correct-operation

Figure 3.19. A later transaction tree for the side effect program

Transactions 10, 11, and 12 can now be completed and removed from the task tree, in that order. Transaction 12 returns the cons of #!true and 'correct-operation in its future. Transaction 6 then returns the result of transaction 12 in its future and is then removed from the tree. This processes is repeated for tasks 4 and 1, until the transaction tree has disappeared. It should be noted that the result of executing the code in figure 3.16 under ParaTran will be a chain of four futures, the last of which contains the actual cons cell returned by the program, presuming the garbage collector has not run during the execution of the sample program.

3.7. I/O Processing

3.7.1 Output

Output is easily handled within the ParaTran model. If each output device is viewed as being a stream onto which outputs are added, then the analogy between output and side effecting writes becomes evident. Each output instruction should just be replaced by code to create a special entry in the write log. Outputs are thereby delayed until the write phase when they are guaranteed to be correct. At this point, the output operations will indeed be performed.

3.7.2 Input

Input is somewhat more problematic than output. Input can be viewed as a stream from which the element at the head can be removed. In order to assure that two different tasks don't both read the same element from the head of a stream, the read and side effect which take place when the head of a stream is removed must be made explicit in the ParaTran model. This can be done by placing an entry in both the read and write logs when an input operation is performed. The result of this approach is that if two tasks each read the same element from the head of a stream, the one which should have executed second on a sequential machine will have its associated task aborted. This will cause the task to be reexecuted; and hopefully, this time the correct input value will be read from the stream.

Obviously, the backup necessitated by repeatedly having to reexecute input tasks could lead to an unacceptable waste of machine time. A potential solution to this problem would be to make input operations a semi-sequentialization point. Input operations would be their own tasks and would not execute, and therefore read an input value, until the input task validated. At this point the given task would be assured not to abort, and the input operation could be done safely. Using this approach the future returned by an input operation could be utilized even before the actual input operation took place. However, this form of input might unnecessarily serialize processing in programs which only do infrequent input operations and would rarely have tasks aborted due to using incorrect input values.

A more general solution to the input problem than just including either of the two types of input presented, or both, is to add a **sequentialize** primitive to the language. This form would be its own task and would cause processing of the argument to sequentialize to be put off until the associated task was the next to complete. In other words, it would allow one to build the second form of input from the first. This is the approach that has been selected in the implementation of ParaTran. It has the virtue that it allows a programmer to control access to data structures which demonstrate similar pathological behavior to that of input streams.

3.8. Error Handling

When a task is executing under ParaTran, there is the potential for it to detect an erroneous error. This is due to the fact that a task can be seeing incorrect bindings between variables and values or fallacious values in side effectable data structures. Such erroneous errors clearly must not be signalled to the user since this would mean that error-free sequential programs could fail to execute correctly under ParaTran. A solution to this deficiency in the basic model is to inhibit the signalling of errors until the write phase. If a task detects an error while running, it should suspend its execution and store any necessary information about the error in some form of task descriptor block. If the task which has detected an error is later aborted, then the error flag in the task descriptor should be cleared. However, if the error still exists following validation, then a real error took place, and it should be signalled to the user.

3.9. Program Termination

The execution model presented thus far has the defect that it does not ensure termination of programs on the multiprocessor which terminated on the uniprocessor. This is a result of the fact that many of the tasks which exist at any given instant might be aborted. If all of the processing power is going to tasks which will be aborted and to creating more tasks which will have a similar fate, the computation which will lead to the abort of this worthless work might never get the processing necessary to initiate the aborts. One way of insuring against this possibility is to place some constraints on which tasks are executing at any given time. The simplest restriction which guarantees termination is to mandate that the next task to be completed is always one of those executing. When it completes its execution, it should immediately have its validate and write phases performed so that at least a little progress is being made. If one processor is always running the next task to be completed, then even if all of the other processors are working on tasks that will be aborted, incremental progress is being made towards finishing execution of the entire program. It is hoped that only very rarely will just 1 of N processors be doing worthwhile computation while the rest are effectively accomplishing nothing. So far, simulations of the ParaTran system support this expectation.

IV. PARATRAN IMPLEMENTATION ISSUES

4.1. Read and Write Logging

4.1.1 Log Record Formats

Read log records have been stated to keep track of queries of variable bindings and object values; and, write log records, of modifications to the same. In order to avoid unnecessary aborts, it would be best if these log records referred to the fields of the specified objects, instead of the entire objects. This would also obviate the need to copy an entire object when only a single field is changed. How best to accomplish this form of logging remains a question for further discussion and investigation. The one requirement that must be met is that it is imperative that log records continue to reference objects correctly in the presence of object motion due to garbage collection. In other words, if two tasks each reference the same field of an object, one before a garbage collection and the other after it, they must both have equivalent read log records. Since garbage collection in Scheme only deals with whole objects, not their individual fields, log records almost certainly must reference the fields of objects by pairs: an object pointer and an offset. If object pointers were regular Scheme pointers, they would automatically be updated at garbage collection time to point to the new locations of the appropriate objects.

4.1.2 Efficiency in Use of Logs

There are several approaches which might be taken when a task makes multiple references to the same variable binding or object field. The straightforward method would be simply to create a read log record every time a read was performed, thereby yielding a read log with potentially multiple entries for the same field of the same object. This means the length of the read log would be determined by the number of references rather than the number of objects referenced. Consequently, a potentially unnecessarily long read log might have to be scanned in order to perform validation of tasks. An alternative technique would be to check the contents of the read log prior to creating each new read log record. This would avoid creation of duplicate read log entries at the cost of having to scan the entire log on each read. Which of these two approaches is actually the most efficient for typical Scheme programs is a difficult question to answer on purely theoretical grounds; however, the latter method is currently used by the ParaTran simulator. There is a similar problem of whether to allow the creation of numerous write log records for the same field of the same object. In this case, the author believes that duplicate entries should not be allowed since every entry corresponds to an eventual broadcast message. Furthermore, the cost of searching the write log on a write should be minimal since rarely will a single task do multiple side effects.

When a read is executed by a task, it must either return the global value of the entity to be read or the result of a side effect performed by this task, as appropriate. If no hardware support is available, this requires searching the entire write log on every read operation to see if the selected item has been side effected. This search could be eliminated if it were declared that a task can do no computation following a side effect. In other words, each task would be limited to doing at most one side effect and would do it as its last operation. This approach would also eradicate the need to check the write log on each write since this log could now only contain a single entry.

4.1.3 Effects of Logging on Memory Consumption

The simple scheme presented in which log records distinguish entities based on an object pointer and an offset has the disadvantage that objects which are no longer accessible by any Scheme code are prevented from being garbage collected because they are still pointed to by the log records. Since defunct objects tend to fill up the heap unnecessarily, it would certainly be desirable to allow them to disappear once the only references left to them are in the logs. This is particularly important if the defunct objects are procedures because the environment in which a procedure is created cannot be garbage collected as long as the procedure exists.

One method of allowing objects to which the only pointers are from log records to be garbage collected is to create **weak pointers**. Weak pointers have precisely the property that they do not inhibit garbage collection. If weak pointers have the additional property of those in MIT Scheme that they are replaced by nil when the object to which they point is garbage collected, then weak pointers cannot be used for both read and write log pointers. This is because the ability to determine if a read and a write log record both refer to the same object is mandatory, even if the object pointed to by the log records is defunct. In order to solve this problem, write log records could use strong pointers; and, read log records, weak pointers. A preferable solution would probably be to have weak pointers to defunct objects be converted to integers having the value of the previous address of the object. Since all weak pointers to an object would be transformed during the same garbage collection and no further weak pointers to this object could be created, consistency amongst references to an object is insured by this approach. The one possible disadvantage to this scheme is that weak pointers to two different objects could be coerced to the same integer if two objects both became defunct while residing at the same address. This type of conflict could lead to an unnecessary abort; but, as will be explained later, ParaTran operates correctly in the presence of such spurious aborts.

4.2. Task Scheduling

4.2.1 Scheduling Considerations

How the scheduling of tasks is done within any system greatly affects its performance. Since the ParaTran approach is intended to be applicable for highly parallel machines, any central resources which can act as bottlenecks must be avoided. As a result, a central work queue is not desirable. The simplest way to avoid this seems to be to have a queue of tasks to be executed per processor. When a processor spawns a task, it places it on its local queue. When a processor needs another piece of work to perform, it first looks in its local queue. If the local queue is not empty, then a task is removed from it based on a local queueing function. If a processor's local queue is empty, then it begins searching all of the other processors' queues, based on a secondary queueing function, looking for work elsewhere in the system. The order in which other processors' queues are searched might often want to be chosen with the topology of an individual system on which ParaTran is implemented in mind.

There are three queueing decisions which as yet remain unspecified. When a processor finishes processing a task, how should it choose the next task to execute from its local queue? If the local queue is empty, how should a processor choose the task to execute from another processor's queue? And finally, when a task spawns a subtask, should the spawner, the spawnee, or some other task be the one which is processed next? The answers to these questions which were utilized in building the ParaTran simulator are far from ideal. They will be described merely as a vehicle for motivating better solutions. The author's best suggestions as to how scheduling should be done as derived from the experience of building and using the simulator are presented following the description of the current approach.

4.2.2 Scheduling in the Simulator

In order to simplify the design of the simulator, only a single work queue was created for all processors to share. This was sufficient to test whether the ParaTran model could work, but has many drawbacks as described in the previous section. In order to satisfy the conditions necessary to guarantee termination, the work queue was divided into two segments: one for priority tasks and the other for all remaining tasks. Priority tasks are those tasks which fulfill two conditions. First of all, they must reside on the leftmost branch of the transaction tree. Leftmost branch tasks are those tasks whose parent is also a leftmost branch task and which are their parent's leftmost child. Furthermore, the root is a leftmost branch task. Since tasks are removed from the system as soon as they complete, the tasks on the leftmost branch of the tree are those which are near the beginning of the sequential completion ordering. The next task to be completed is always a member of this set.

The second condition for membership in the set of priority tasks is that it must be possible to ensure that a priority task will never be aborted. This condition is equivalent to saying that an application task which is a leftmost branch task must be validated by the normal completion processing before either it or its children enter the set of priority tasks. Once an application task is validated, it becomes a member of the set of priority tasks, along with its appropriate descendants. By having free processors always choose to execute priority tasks, if any are on the queue from which these processors are acquiring work, the requirements necessary to ensure termination are met. If leftmost branch tasks which might still abort were allowed in the priority task set, then termination would not be guaranteed since all of the processing power might still be going to tasks which should eventually be aborted.

The queueing function used by the simulator to select the next task to execute when a processor becomes free is basically FIFO. Tasks are deposited and removed from each segment of the queue in a FIFO fashion. When a new task is needed, first the priority task portion of the queue is checked for an available task; and then, if none exists, the other segment of the queue is inspected. It should be noted that tasks sometimes must be moved from the general section of the queue to the priority portion, since as tasks are completed and removed from the system, new tasks become members of the priority set. The second queueing policy is that when a task is spawned, the spawner continues executing and the spawnee is placed on the work queue.

The queueing functions which were selected for the simulator may conceivably have been the worst combination possible. In order to get some idea of the implications of a queueing mechanism, it can be instructive to look at its results on a uniprocessor machine. A task to evaluate a function call will eventually spawn subtasks to evaluate the operator, each of the operands, and to perform the actual function application. Using the choice of continuing execution of the spawner, all of these tasks are spawned before any of them is processed. Once they have all been spawned, a similar form of task creation will take place for the function calls in each of the subtasks. In essence, a breadth-first search of the computation tree has been induced. This means that the task tree grows very large at runtime, using up much of the heap space. The population explosion expected from breadth-first search has been seen in using the ParaTran simulator.

Furthermore, since all tasks must be completed before a program can terminate, it would seem desirable, whenever possible, to be executing tasks which appear near the front of the sequential completion ordering. The decision to processes the spawner, rather than the spawnee, tends to cause the exact opposite to take place. While giving priority to tasks which will complete earlier might sometimes be in conflict with the desire to be executing tasks in parallel which do not interact through side effects, and consequently can't cause each other to abort, it is in general the correct idea.

4.2.3 Improved Scheduling Ideas

It is the belief of the author that the use of a segmented LIFO queue in conjunction with a spawnee executes first scheme would greatly improve the performance of ParaTran. These decisions would tend to induce a depth-first execution of the task tree, limiting the proliferation of tasks and keeping the computation near the beginning of the completion order. It should be noted that on a uniprocessor, these queueing decisions lead to the same order of execution that would result from traditional, sequential execution of Scheme. The one potential problem with this approach is that it might lead to tasks running in an order that causes many aborts. If this is the case, the author suspects that it will result from multiple subexpressions of the same sequences performing their read phases before the earlier subexpressions have had a chance to complete. This is because the only effects that any of the subexpressions of a sequence can have, except for the last, are through side effects. It has not yet been possible to determine with the simulator whether this is a serious difficulty; however, it might in the future be necessary to investigate more sophisticated schedulers to alleviate this problem.

4.3. Message Broadcasting

In the original ParaTran description it was stated that write messages are broadcast to all other tasks by a given task when it does its writes. How to do this efficiently is still somewhat of an open question. Explicitly sending messages to every task which has yet to be completed seems to be a very bad approach. Instead one might consider keeping a list of all messages broadcast. Each task could have a pointer into this list, corresponding to the time at which the task was created, and all messages broadcast since that time could be found by chaining down the list from that point. Tasks would receive messages by periodically checking to see if any new ones had been added to the list. As long as the entire list was checked by the end of the validation phase of any given task, this approach would suffice to guarantee correct operation. However, the broadcast list would be a central resource on which contention might result. A possible solution to this would be to have a broadcast list on every processor. Now, each time a task wanted to broadcast a write, it would send a message to every processor instead of every process. This is obviously much more efficient, and such global broadcasts might even be explicitly supported by the interprocessor communication network. Tasks would now have pointers into the broadcast lists on the processors on which they were created. Everything else would behave in the same manner as for the single broadcast list case.

The broadcast list approach presented leads to tasks having to process many messages which are assured to be irrelevant to the given tasks. These are writes which are broadcast between the time a task is created and the time at which it actually begins executing. Since a task has yet to read any values before it begins executing, it certainly can't need to be aborted due to one of the broadcast messages sent during this time period. In fact, if a task waits too long to read its messages, it might actually read a location about which it has been sent an unprocessed broadcast message and eventually abort unnecessarily. These inefficiencies could be solved by creating the pointer into a broadcast list at the time a task starts executing. In this way only those messages broadcast after the point at which execution begins would be processed by a given task. This approach has the added advantage that a task could be made to point into the broadcast list of the processor on which it begins execution, as opposed to the list on the processor on which it was created. This should lead to most references to the broadcast list being local to the processor on which a task is executing. As will be described in the chapter on architectural support, this can lead to significant performance advantages.

4.4. Compilation for ParaTran

Through the use of a somewhat sophisticated compiler, more efficient code could be generated for execution by ParaTran. The improvements would come in two major areas: the granularity of tasks could be increased and not all variable reads need to be logged. Both classes of optimizations are based on distinguishing identifiers which cannot be rebound during the execution of a program. Read log records are only useful for detecting the reading of a value by a task before a task which should have executed first in the sequential model has had a chance to update the value. If no task can ever update a given variable binding, then recording reads of that binding is clearly superfluous. In Scheme there seem to be a very large set of variables whose bindings can never be modified. They are used solely for the purpose of having a name by which to identify arguments which are passed to a function or intermediate results. By dividing variable references into two classes, based on whether the bindings are modifiable, almost all logging of variable reads can be eliminated. (The current MIT Scheme compiler does this type of analysis on variables.)

In order to increase task size it is important to know that the meanings of the language primitives are the ones initially built into the system. This cannot be guaranteed in general since the user can rebind any identifier in Scheme, even to the extent of redefining what '+' or 'car' mean. Since just analyzing an individual user program is not enough to assure that the primitives have not been changed, a declaration to this effect is necessary if the compiler is to make use of this information. Such declarations are common practice when one wants to do efficient compilation of Lisp like languages. Once the primitives are known to be unchangeable, all of the user functions which compose a program can be classified based on a number of criteria. Functions which only depend on functional operations can be identified. All tasks into which such functions are decomposed needn't perform their write phase.

All of the subtasks of a task can be eliminated by amalgamating them into the parent task if none of the subtasks themself spawn subsubtasks. This means that by starting at the leaves of a computation tree, which are all calls to primitives, larger tasks can be composed. The restriction that no subtasks which are joined into the parent task can themselves spawn subsubtasks is necessary to ensure the correct sequential result. If a task did some computation, then spawned a child, and then did some more computation, neither completing the child first and then the parent nor using the reverse completion ordering would yield the correct result. This is because all of the computation performed by the parent would effectively be sequentialized either before or after that of the child.

As an extension of the above procedure, the evaluation of the operator and the operands of a function application can often be made part of the application task. Since an application task is always validated prior to the validation of its subtasks, the evaluation of operator and operand tasks can be made a part of the function application task if the former do not spawn any subtasks or do any side effects. If the write phase of application tasks is also made to take place before the validation of any subtasks, then the restriction to side effect free operators and operands can be removed. Furthermore, by taking advantage of the fact that the semantics of Scheme do not specify the order of evaluation of the operator and operands, only that the order is fixed for all executions of the same piece of code, any subset of the operator and operand tasks which meet the specified conditions can be made a part of the application task. This has the effect of choosing an evaluation order in which operands and operators which require subtasks for their evaluation will be done first and all others will be done afterward. Obviously, the granularity of task size should only be increased to the point at which it would become productive to enable two processors each to be doing portions of some task in parallel. The crossover point for this trade-off will obviously be dependent on the overhead inherent in any given implementation of the ParaTran runtime system.

4.5. Optional Task Spawns

A classic source of inefficiency in multiprocessing systems is the creation of too many tasks. The spawning of each task requires some amount of processor time; and, the storing of each task, some amount of memory. Ideally, one would like to have just enough tasks present in a system so that there is exactly one available any time a processor is through with its current task. Any additional tasks on the work queues not needed to meet this goal are a drain on a system. One way to attempt to achieve this delicate balance is through optional task spawns. When a system is about to spawn a task, it could check to see how many tasks are already available on the work queues. If there were a large number of tasks present, then a system could either ignore the task spawn and run the new task as a portion of another task or postpone doing the task spawn until some of the already available tasks have been utilized.

Optional task spawns can be added to ParaTran given a correct set of implementation decisions. The runtime mechanism presented thus far utilizes two varieties of tasks: application tasks and all other tasks. The only difference between these two types of tasks is that the former are validated before their children and the latter are either leaf tasks or don't do any computation. This means that all tasks could be replaced by application tasks. If one chooses to do the write phase of application tasks before their children are completed, which has previously been stated to be a viable design decision, optional task spawns become possible. Based on the same reasoning that allows the operator and operand subtasks of a function application to be incorporated into their parent task at compile time, any task spawn can be ignored at runtime, and the subtask run as a portion of its parent, so long as its parent task has not already spawned any children. This restriction is necessary since the parent task will now be completed prior to any of its children; and, if a task spawn were eliminated after another child were already spawned, this would have the affect of reversing the sequential completion ordering of the two subtasks.

As a result of the above observations, all that is necessary to implement optional task spawns is a single bit in each task descriptor which tells whether the given task has already spawned any subtasks. The task spawn mechanism could be altered so that whenever a spawn were encountered, the size of the work queues could be checked to see if ignoring the spawn would be desirable. If removing the spawn would be advantageous, then the new bit could be examined to make the ultimate decision as to whether to spawn a new task. In this way the efficiency of the ParaTran could, in the author's opinion, be greatly improved. If a work queue per processor were being used then, the spawn decision should most probably be made based on the number of entries in a processor's individual queue. This is because checking the amount of work on all the separate queues would undoubtedly be too inefficient to be practical.

4.6. Tail Recursion for Reductions

A **reduction** is a situation in which a function calls another function in such a manner that the caller will return whatever result is returned by the callee. In figure 4.1 one sees an example of a function, decrement, which reduces into the subtraction function. Similarly, as has been stated earlier, sequences always reduce into their last subexpression; and, conditionals, into their consequent or alternative.

```
(define (decrement n)
  (- n 1))
```

Figure 4.1. Decrement tail recurses into subtraction

A **tail recursive** interpreter takes advantage of its knowledge of reduction in order to limit the amount of stack space required to execute a program. When a reduction is performed, the only portion of the stack frame of the caller which is necessary following the call is the pointer to the piece of code to which control should return following the completion of the execution of the callee. None of the values of the local variables or arguments to the caller are of any importance. Therefore, the stack frame of the caller can be eliminated if the return pointer of the callee is made to be the return pointer of the caller. This approach not only saves stack space, but also means that returning from a deeply nested set of reductions costs constant time rather than time proportional to the depth of the recursion.

The ParaTran runtime system does not currently incorporate tail recursion in its implementation. This is as a result of the fact that the task tree grows with each recursive call, even though the stack doesn't need to. Consequently, deep recursion can lead to running out of space in the heap. This is particularly problematic since Scheme does not have an iteration construct; and therefore, recursion is used in its place. This is done with the expectation that tail recursion will cause programs written in a recursive style to execute efficiently. As an example, the top level read-eval-print loop of the MIT Scheme interpreter which reads user commands and executes them is implemented as a recursive function. This function reads a single line of input, evaluates it, and then calls itself to process the next input. This design depends on the assumption that no stack or heap space will be used up by each recursive call.

Tail recursive shrinking of the task tree can be added to ParaTran if all tasks are treated as application tasks and are completed before their children, as was suggested in the previous section. In this case, a parent task can be removed from the task tree and replaced by its child once it has completed and it only has a single child remaining. This state exists once all of the children of the parent task except for one have completed and been removed from the task tree. A degenerate case is that as soon as a parent task which has only spawned one child completes it can be removed. Adding this form of tail recursion would cause the transaction tree in figure 3.19 not to include tasks 1 and 4.

4.7. Distributed Garbage Collection

Since ParaTran is based on a shared heap model in which any processor can manipulate any Scheme object in the system, some form of distributed garbage collection is required. How to do garbage collection efficiently on a multiprocessor computer is an as yet unsolved problem and the subject of much ongoing research. Consequently, when ParaTran is eventually implemented on a multiprocessor, the author intends to avoid this complication altogether by using the results of the work of others. In particular, a multiprocessor garbage collector which has been designed for the same Scheme system being utilized for current ParaTran research [3] will undoubtedly be used.

There is one area in which garbage collection for ParaTran could be made to take special advantage of its unique approach to parallel execution. The ParaTran model can tolerate spurious aborts of any transactions which have yet to be completed. This means that if the garbage collector finds that there is just not enough free space in the heap for efficient execution, it can abort some tasks. This would cause all subtasks of the aborted tasks to be killed and would free part of the heap. The trade-off between throwing away some work and doing more frequent garbage collection would undoubtedly be an interesting one to investigate. If nothing else, the abort-when-full approach allows a way out when the heap actually becomes completely full.

4.8. Debugging

Unlike most multiprocessing languages, debugging Scheme as executed by ParaTran is no more difficult than debugging a standard uniprocessor version of the language. This is a result of users having a strictly sequential view of the ParaTran system.

The author believes this is a distinct advantage of ParaTran over other approaches to multiprocessing. One of the most difficult classes of bugs to find are those which appear inconsistently. Bugs resulting from the multiprocessing nature of programs are inherently of this type and often require monumental effort to detect and correct. By avoiding these bugs altogether, code run under ParaTran is made considerably simpler to debug.

V. THE PARATRAN SIMULATOR

A simulator of the ParaTran system has been built by the author in order to investigate some of the properties of this approach to parallel execution. The simulator takes the number of processors to simulate as a parameter and then time slices amongst that many virtual processors. The pieces of information which are returned from a simulation are the number of tasks spawned during the simulation, the number of those which were aborted and killed, the number of variable read and writes logged, and the number of logs of reads and writes to other data structures which were kept. Also, the average level of processor utilization is returned. This simulator makes no attempt to estimate the overhead of the ParaTran scheme in terms of actual execution time. To do so would be impractical since both the simulator and the underlying ParaTran mechanism are programmed in Scheme; and consequently, this implementation has an astronomical overhead. The purpose of this simulator is to give the author a feel for the types of overhead which would be incurred in a true multiprocessor version.

Due to the speed of simulation on the available hardware, only fairly small simulations have been possible. In addition, an unfortunate choice of queueing functions has limited the ability to run large simulations. As a result, the author feels that it is unwise to attempt to read much into the simulation results. Until ParaTran is implemented on some real multiprocessor or a simulator which can handle programs containing hundreds of lines of code is available, no definitive conclusions can be made.

The results obtained so far have been those which one would naively expect. Functional programs cause no aborts or kills; and, programs which use side effects cause both. In particular, the program in figure 3.16 whose task tree is shown in figure 3.18 has under simulation aborted task 6 and killed tasks 7, 8, and 9 just as was theorized as a possibility in a previous discussion. No determination of the level of aborts and kills in any real programs has been feasible.

An interesting observation the author has been able to make from using the simulator is that it is very rare to get chained aborts. These result when a task which has already returned a value aborts, causing any tasks which have already utilized the returned value to abort, etc. Chained aborts are one of the possible motivations for wanting to use a true nested transaction model, and their absence is very reassuring. However, further exploration is definitely necessary.

VI. PARATRAN ARCHITECTURAL SUPPORT

As mentioned in the introduction, ParaTran would like to view the heap as a single uniform object. This makes hardware support for a shared memory very desirable. For systems incorporating thousands of processors, having a single memory unit networked to all of the processors can be very expensive in terms of access time. Therefore, it is probably desirable to have the memory partitioned into one piece per processor. Each piece of memory would be tightly coupled to a single processor, but accessible to other processors over some interconnection network. In order to make accesses to any portion of memory both efficient and transparent, an I/O coprocessor could be utilized at each node of the machine. This processor could cause local references to go to the local memory, and nonlocal references to be converted into requests for the appropriate I/O processor to handle the reference. Such requests would obviously be communicated via the interconnection network. Utilization of an I/O coprocessor would save the main processor the overhead of having to control communications over the network and most likely improve performance.

Futures support would also be quite valuable. If the value of a future object is needed, it seems very wasteful for a processor to have to read the future object first and then have to read its value. This is particularly true if the future object and the value both happen to be in the local memory of another processor, necessitating network traffic for each of the two accesses. A possible remedy for this situation is to educate the I/O processors about futures. Two classes of reads could be created: the old-fashioned variety and one which chains through futures. If a chaining read were requested and the I/O processor found the value read to be a keep-slot or determined future, it could automatically issue a request to read the future's value. If the future were undetermined then the I/O processor could return a message to this effect. The local I/O coprocessor could then issue a trap to its corresponding processor, causing the code to initiate waiting on a future to be invoked. Given such an architecture, the use of futures would become almost entirely transparent. The one consideration which has yet to be given attention in this approach is the need to log the reading of keep-slots. One solution might be to return a list of keep slots chained through along with the data item requested. The local I/O processor could then issue a trap to log the keep-slot references, followed by returning the actual data value. A more complete solution would be to also support logging in hardware.

The I/O coprocessor could support the logging of reads and writes by creating two varieties of each I/O operation: one with logging and the other without. Reads and writes without logging would be the conventional ones previously supported. A read with logging would return the requested data item, but would also add a log record for the item read to a list in the coprocessor. A write with logging would merely create a write log record in the coprocessor. The write log could now act as a type of cache so that reads would return the appropriate value: either the one buffered in the write log or the one in the actual location. Additional operations would be required to clear the logs, to dump them into the heap in order to swap out a task, and to reload them from the heap when a partially executed task's processing was to be continued. Also, the logging portion of the coprocessor could be used to do the actual updates necessitated during the write phase of a transaction and to broadcast information about those updates to the other transactions. Similarly, the coprocessors could process the received broadcast messages and signal the possible need to abort a currently executing task.

Since nonlocal memory references might take quite a number of clock cycles, 'micro task swapping' is potentially quite valuable. By having multiple register sets and internal state variables, a processor can be designed which can swap between tasks between one machine cycle and the next. Such a processor would be useful in a multiprocessing environment in order to enable execution of a second task to take place while an initial task is waiting for a nonlocal memory reference to be completed. In this case, a nonlocal reference would become very much like a page fault, with the I/O processor issuing a trap to cause a micro task swap to take place. If logging support were also incorporated into the I/O coprocessor, then duplicate logs would be required, one for each micro task.

VII. CONCLUSION

The ParaTran simulator demonstrates that it is possible to use the ParaTran approach to correctly execute Scheme programs in parallel. These results are further supported by those derived from a project called Time Warp [6], which is a system for doing event based simulations based on ideas of optimistic concurrency similar to those used in ParaTran. These successes will hopefully motivate others to investigate this novel approach to parallel processing. The next step in the investigation of ParaTran would be to build a more efficient simulator, incorporating some of the improvements in scheduling of tasks and logging of reads and writes, as well as possibly others. Such a simulator would give one a better feeling for the expected performance of a truly parallel implementation of ParaTran. Following further simulations, a truly parallel ParaTran system, and eventually special purpose hardware, would be desirable.

A. SCHEME PRIMER

This appendix contains a brief overview of those features of Scheme with which one must be familiar in order to understand the descriptions in the text of this thesis. Many of the descriptions given are borrowed in whole or in part from "The Revised Revised Report on Scheme or An UnCommon Lisp" [2]. The legal identifiers for the purposes of this explanation are any string of characters composed of letters, asterisks (*), and dashes (-). All variable scoping is strictly lexical. Unless a form in the function position is specified to be a special form in the descriptions that follow, the first term in a set of parenthesis is assumed to be a function and it is applied to the results of evaluating the other terms in the parenthesis, which are assumed to be arguments.

(quote *datum*) or '*datum*

A special form which evaluates to *datum*. This notation is used to include literal constants in Scheme code.

```
        (quote a)                 -->  a
        'a                        -->  a
        '(+ 1 2)                  -->  (+ 1 2)
```

(- *z1 z2*)

Returns the result of taking the difference of *z1* and *z2*.

```
        (- 3 4)                   -->  -1
```

(-1+ *z*)

Returns the result of subtracting 1 from *z*.

```
        (-1+ 5)                   -->  4
```

(cons *obj1 obj2*)

Returns a newly allocated pair whose car (first component) is *obj1* and whose cdr (second component) is *obj2*.

```
        (cons 'a 'b)              -->  (a b)
        (cons 'c 1)               -->  (c 1)
```

(car *pair*)

Returns the contents of the car field of *pair*.

```
        (car (cons 'a 'b))          -->  a
```

(cdr *pair*)

Returns the contents of the cdr field of *pair*.

```
        (cdr (cons 'a 'b))          -->  b
```

(sequence *expr1 expr2* ...)

A special form which evaluates the *exprs* sequentially from left to right and returns the value of the last *expr*.

```
        (sequence (-1+ 3)
                  (cons 'c 'd))     -->  (c d)
```

(lambda (*var1* ...) *expr1* ...)

Each *var* must be an identifier. The lambda expression is a special form which evaluates to a procedure with formal argument list (*var1* ...) and procedure body (sequence *expr1* ...). The environment in effect when the lambda expression was evaluated is remembered as a part of the procedure. When the procedure is later called with some actual arguments, the environment in which the lambda expression was evaluated will be extended by binding the identifiers in the formal argument list to fresh locations, the corresponding actual argument values will be stored in those locations, and (sequence *expr1* ...) will then be evaluated in the extended environment. The result of (sequence *expr1* ...) will be returned as the result of the procedure call.

```
        ((lambda (x) (- x 3)) 4)    -->  1
```

(define (*func var1* ...) *expr*)

A special form which extends the current environment to contain *func* and binds this identifier to (lambda (*var1* ...) *expr*).

```
        (define (dec x) (-1+ x))    -->  dec
        (dec 12)                    -->  11
```

(set! *var expr*)

A special form which evaluates *expr* and then changes the binding of *var* to be the result of having evaluated *expr*.

```
(set! a 5)                    -->  5
a                             -->  5
(set! a (- a 2))              -->  3
a                             -->  3
```

(let* ((*var1 form1*) ...) *expr1* ...)

A special form which creates a new lexical environment in which to evaluate the *exprs* and then evaluates them in this environment. The new environment is built by successively creating a slot for each of the *vars* (left to right) and binding it to result of evaluating the associated *form*. The result of the last *expr* is returned as the result of the let*.

```
(let* ((a 5) (b 3))
  (- b a))                    -->  -2
```

#!false

The unique false object.

#!true

A unique true object.

(if *condition consequent alternative*)

A special form which first evaluates *condition*. If it yields a true value (any object other than the false object), then *consequent* is evaluated and its value is returned; otherwise, *alternative* is evaluated and its value is returned.

```
(if #!false 'a 1)             -->  1
```

(zero? *var*)

Returns true if *var* has the value 0 and false otherwise.

```
(let* ((a 0))
  (if (zero? a)
      'true
      'false))                -->  true
```

B. HEAP STORAGE FOR SCHEME

The following is a description of a typical approach to heap storage for the Lisp family of languages. It is based on the method used by MIT Scheme, the language in which the ParaTran simulator is implemented.

All of the words in the Scheme heap are typed. They are composed of a type code and a value field. Simple objects like fixed point numbers, floating point numbers, and characters are stored in a single word with the appropriate type code. More complex objects require multiple words of storage. Cons cells (pairs) are a data structure which is just a two element record. They are formed from two consecutive Scheme words. A cons cell is stored in some other structure by placing a pointer to (i.e. the address of) the first of the consecutive words of the cons cell in that structure. The pointer would have type code "cons cell" so that the system would know what type of object is being pointed to. Similarly, vectors are stored as a series of Scheme words, the first of which is a fixed point number specifying the length of the vector and the rest of which are the actual vector entries. A vector can be stored in another structure by storing a pointer to the first word of the vector, with the type code "vector", in that structure.

Box-and-pointer notation [1] is simply a diagrammatical way of representing the storage scheme explained above. Each box in this notation contains a type code on the left and a value on the right. Consecutive words of storage are represented by a series of boxes in a vertical stack. Arrows are used for pointers, instead of addresses, in order to increase readability. In this way, a visual presentation of the way in which a series of structures are interrelated can easily be represented.

References

Abelson, H., and G. Sussman, *Structure and Interpretation of Computer Programs*, M.I.T. Press, Cambridge, Mass., 1984.

2. Abelson, H., *et al.*, "The Revised Revised Report on Scheme or An Uncommon Lisp," M.I.T. Artificial Intelligence Laboratory Memo 848, Cambridge, Mass., Aug. 1985

3. Courtemanche, Anthony, *MultiTrash, a Parallel Garbage Collector for MultiScheme*, BS thesis, Massachusetts Institute of Technology, Jan. 1986.

4. Halstead, R., "Implementation of Multilisp: Lisp on a Multiprocessor", *ACM Symposium on LISP and Functional Programming*, Austin, TX, August 1984, pp. 293-298.

5. Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems* 7:4, Oct. 1985, pp. 501-538.

6. Jefferson, David R., "Virtual Time", *ACM Transactions on Programming Languages and Systems* 7:3, July 1985, pp. 404-425.

7. Kuck, D. J., R. H. Kuhn, B. Leasure, and M. Wolfe, "Analysis and Transformation of Programs for Parallel Computation," *Proceedings of the Forth Inernational Computer Software and Applications Conference*, Oct. 1980.

8. Kung, H. T. and Robinson, John T., "On optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems* 6:2, June 1981, pp. 213-226.

9. Moss, J. Eliot B., *Nested Transactions: An Approach to Reliable Distributed Computing*, M.I.T. Laboratory for Computer Science Technical Report TR-260, Cambridge, Mass., April 1981.

10. Pyle, I. C., *The Ada Programming Language*, Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1981.

11. Reed, David P., *Naming and Synchronization in a Decentralized Computer System*, M.I.T. Laboratory of Computer Science Technical Report TR-205, Cambridge, Mass., Sept. 1978.

ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme

Prepared by:

Morry Katz

Rockwell International Corporation

Science Center

1049 Camino Dos Rios

Thousand Oaks, CA 91360

July 1987

ABSTRACT

The number of applications requiring high speed symbolic computation and the performance requirements of these projects are both rapidly increasing. However, the computer science community's ability to produce high performance uniprocessor hardware is being outstripped by these needs. Therefore, we propose a unique multiprocessing solution to the high speed, symbolic computation problem. Our approach is to develop a transparent runtime mechanism for executing standard, sequential Lisp code on a multiprocessor computer. ParaTran, as we call our system, is based on the concept of atomic transactions as developed for use in distributed database systems, programming languages, and operating systems. It utilizes an optimistic scheduling algorithm for processing transactions in order to maximize the available parallelism. In this way we believe that we can create a system which is both easy to use and yields exceptional performance.

Our concept is based on dividing a Lisp program into a series of pieces, or transactions, which have an a priori sequential order in which they would be executed on a uniprocessor machine. However, instead of performing this serial process, we optimistically run multiple transactions in parallel and then detect at runtime when this parallel execution is not "serializable". An execution ordering is not serializable if it leads to a different result than that which would have arisen from sequential processing. When such conflicts are detected, ParaTran will reexecute certain transactions in order to effectively serialize the computation. Similar approaches using optimistic concurrency have been quite successfully exploited by developers of distributed database [Kung81] and discrete-event simulation systems [Jeffe85].

INTRODUCTION

Many of the systems of the future, both within and outside the defense sector, will require high speed symbolic computation. *Expert Systems*, *Robotics*, and *Artificial Intelligence* are just a few of the areas in which this need has already arisen. At the present time, one of the most promising methods for meeting this demand appears to be through the utilization of parallel processing. In recent years, hardware technology has progressed to the point where a number of tightly-coupled multiprocessing computers have been developed (e.g. the BBN Butterfly, the Caltech/JPL Hypercube, etc.). However, programming these machines remains an extremely difficult task and the development of appropriate programming techniques and languages continues to be an open research question.

One possible solution to the parallel programming problem is the use of a form of optimizing compiler to translate sequential programs into parallel implementations. Typically such compilers form data dependency graphs in an attempt to identify independent threads of computation which can be executed concurrently on different processors. This form of analysis has been most successful in identifying cases in which multiple iterations of a loop can actually be performed in parallel. Professor Kuck at the University of Illinois has done considerable work in the area of parallelizing compilers [Kuck80]. Although his techniques are quite successful for parallelizing Fortran programs, their applicability to symbolic languages seems fairly limited. This results from the relative simplicity of the semantic model of Fortran in comparison with that of symbolic languages like Lisp. In fact, many researchers believe that some of the features of Lisp such as first-class procedures, objects, and continuations, the late binding of function names to code, and the incremental development model make extensive use of compile time detection of parallelism in Lisp impractical.

Another potential solution to the symbolic, parallel programming problem is the use of languages which contain explicit parallelism constructs. Three examples of this form of language are Qlambda developed by Gabriel and McCarthy at Stanford University [Gabri84], Multilisp developed by Bert Halstead at the Massachusetts Institute of Technology [Halst84, Halst85], and Butterfly Commonlisp developed at Bolt, Beranek, and Newman (actually an offshoot of Multilisp) [Stein86, Scott86]. A common characteristic of all of these languages is that they are imperative. Imperative languages allow side effects, or mutations, to be performed on variable bindings and data structures. All explicit parallelism languages which are imperative suffer from a unique set of difficulties of which synchronization and repeatability are two representatives. (Functional languages will be discussed separately.) It is often very difficult for a programmer to partition a single problem into a number of parallel threads of execution which do not interact in unexpected ways through side effects. In practice, this program development problem also manifests itself at debug time in the form of nondeterministic bugs resulting from incorrect decomposition. Due to their sporadic nature, such bugs are exceedingly problematic since they can lay dormant in a piece of code for months or years before they manifest themselves. The emergence of such bugs can even be prompted by hardware modification, such as increasing the number of processing units, or a marginal skew in clock rate.

Explicit parallelism languages also tend to suffer from a lack of effective procedural abstraction. This is because the programmer must know not only what a piece of code does, but how it does it, in order to determine if it can be run in parallel with some other piece of code. For example, one implementation of a procedure to add an entry to a symbol table might be capable of being run in parallel with other invocations of itself, while a second implementation might not. On a sequential machine these two implementations are equivalent, but on a parallel machine they are quite different. A programmer using an explicit parallelism language must insure that two invocations of a procedure which cannot be run in parallel with itself are never inadvertently executed in this manner. This entails guaranteeing that two procedures both of which call a procedure of this type are never run concurrently. Similarly, procedures which call these two procedures cannot be run in parallel, etc. This causes a terrible software engineering problem, particularly for the building of large systems by teams. Worse yet are the software maintenance difficulties which result. A localized knowledge of a piece of a large software package is no longer sufficient to determine the effect of a small modification to a single piece of a program.

A third potential solution to the parallel programming problem is the use of totally functional languages. Although a substantial amount of research has been done in this area, as yet no one has presented a completely satisfactory method of working around the more restrictive semantics of functional languages in comparison with their imperative alternatives. Despite attempts to create better tools for aiding in the development of functional programs, there remain numerous programming constructs and applications which just do not seem to have simple or convenient functional representations (section 3.1 or [Abels84]). Furthermore, there are some classes of problems which don't appear to be expressible through functional semantics. Until these software engineering problems are solved, parallel implementations of functional languages will not become a viable alternative for building large symbolic applications.

Therefore, we offer a fourth, largely uninvestigated, solution to the symbolic, parallel programming problem. We believe that it is superior to the aforementioned methods of parallelization because the transparency of our approach shields the programmer from the details of parallel execution. This yields the potential to take advantage of parallelism which is largely unexploited and unexploitable by the alternatives. ParaTran is completely transparent to the programmer in that it is designed to execute standard, sequential Lisp code on a multiprocessor. In other words, ParaTran makes a multiprocessor appear to the user to be a very high speed uniprocessor engine. Through this approach we feel that we can avoid many of the software engineering problems encountered by users of explicit parallelism languages, while retaining the more powerful semantic model of imperative languages.

The ParaTran model of computation is based on a runtime analysis of parallelism. By postponing parallel execution decisions until runtime, they can be made on a data dependent basis. This gives our system a distinct advantage over most systems using strictly compile time analysis and explicit parallelism languages for which the execution method for a piece of code is typically fixed for all input data. It is very often the case that two pieces of code can be executed in parallel 99% of the time; but, 1% of the time the input data precludes parallel execution. A prime example of this would be a procedure which adds an element to a set. This procedure might be implemented in such a way that two set additions can be performed at once as long as no set is having two elements added to it concurrently. A flow analysis compiler for Lisp can not in general determine if two variables are both bound to the same object. Therefore, in most instances a compile time parallelization system would have to insure that the aforementioned procedure was never executed in parallel with a second instance of itself, thereby sacrificing potential parallelism. Similarly, a programmer using an explicit parallelism language would experience great difficulty assuring that only correct invocations of the above procedure would be run in parallel. In all likelihood, the resulting program would either not take advantage of all of the parallelism or, worse yet, contain an insidious bug.

TECHNICAL APPROACH

ParaTran's unique runtime approach to parallelism is based on a number of state-of-the-art concepts from several different areas of computer science. From concurrent and distributed databases, systems, and programming languages, it borrows the concepts of **atomic** and **nested transactions** [Lisko83, Moss81, Reed79]. To this it adds parts of two approaches to **optimistic concurrency**: one for distributed databases [Kung81] and one for distributed, discrete-event simulation [Jeffe85]. These ideas are then combined and modified for applicability to the domain of parallel programming languages.

This description of ParaTran will begin with an overview of atomic and nested transactions and their uses. The different methods of achieving optimistic concurrency will then be presented. Finally, we will explain how these concepts can be applied to create a parallel implementation of Lisp which appears to the user to be sequential.

The atomic transaction is most easily explained by describing its use in implementing distributed database systems. As used in this context, the atomic transaction consists of a set of operations to be performed on a database in such a manner that the entire transaction appears to take place as a single operation. In other words, an atomic transaction takes a database from one consistent state to another. The only two states of the database which are visible to other transactions are those which existed before the given transaction was initiated, and after it was completed. An example of the utility of atomic transactions would be an attempt to transfer money between two accounts in a bank database. Such a transaction is composed of two operations: the removal of the funds from the first account and their deposit into the second. The database should never be visible in a state in which the transferred funds are in neither account, or in both. Therefore, the transfer must be atomic. Once one has implemented a transaction based system, it is desirable to be able to build a hierarchy of transactions, or nested transactions as they are called. A group of transactions can be combined into a single atomic unit by nesting them as children of a single parent transaction. Not only are the subtransactions of the parent supposed to execute atomically with respect to each other; but, the entire group should appear to happen atomically as viewed by other transactions. Given the ability to nest transactions one level deep, there is no reason not to generalize this to arbitrary nesting of transactions.

In order to maximize the amount of concurrency utilized by a transaction based system, the transactions can be scheduled in an "optimistic" fashion. Rather than using locks to guarantee the apparent sequentiality of a transaction system, optimistic scheduling allows transactions to be run in parallel before it can be assured that the concurrent execution will yield the correct sequential result. If it turns out that parallel execution was not possible, this fact is detected based on information recorded about the transactions, and some form of rectification and reexecution of code is required. This approach to scheduling is desirable for two reasons. Most importantly, it is often impossible even at runtime to determine if parallel execution is possible until it is attempted; therefore, an optimistic scheduler yields greater concurrency. Secondly, it can actually be more efficient to detect the violation of the sequential model and take appropriate action at that time than it would have been to decide if parallel execution was possible in the first place. However, there are obviously costs involved in this form of execution. State information about the history of transactions must be retained to enable **roll back**, and the roll back itself will require some processor time.

Kung and Robinson of Carnegie-Mellon University developed an optimistic scheduling algorithm for atomic transactions (without nesting) as the basis of a distributed database system [Kung81]. In their system, the execution of a transaction is divided into three phases: **read**, **validate**, and **write**. Conceptually these phases are performed as follows. During the read phase, a dry run of the transaction is performed. All database entries which are read during this phase are recorded on a **read list**; and, all updates which would have been made by the transaction had it not been a dry run are recorded on a **write list**. During the validate phase, each entry in a transaction's read list is checked to determine whether the value read during the read phase is the same as the value currently stored in the database. If the values differ, then the transaction was incorrectly executed concurrently with some other transaction. Therefore, the transaction which was validating must be restarted and its read phase repeated. Otherwise, the transaction succeeds in validating, and during its write phase, the updates specified in its write list are applied to the database. As long as no two transactions are performing their validate or write phases at the same time and all transactions validate and write phases are performed in the sequential order in which they are to appear to be executing, this system is guaranteed to give the same result as a sequential implementation. The Kung and Robinson system can be generalized to handle nested transactions in a fairly straightforward manner [Katz86].

Time Warp, a system for performing distributed, discrete-event simulations, developed by Jefferson, currently a Professor at UCLA, and Sowizral, currently a researcher at the Schlumberger Palo Alto Research lab, utilizes a different form of optimistic concurrency [Jeffe85]. The Time Warp model of computation is based on objects and message passing. Each message is sent with a time stamp which is explicitly specified by the program/programmer. These time stamps determine the order in which the objects should appear to send and process their messages during the simulation. However, Time Warp performs optimistic message processing in that all objects are allowed to evolve in parallel by processing those messages they have already received, in time stamp order. Occasionally, this means that an object will receive a message with a time stamp which is earlier in the temporal ordering than the time stamp of another message to the same object which has already been processed. In this case, the object which has just received the message to have been processed in its past must be rolled back to a time at or preceding the time of the just arrived message. Once this has been done, all messages from that roll back time on must be reprocessed. Roll back involves two parts. The first is a restoration of the state of the object to that which existed at the roll back time. The second is the retraction of any messages initiated by the roll back object subsequent to the roll back time. Messages are retracted in Time Warp by sending what is called an **antimessage**. As one would expect, if an object receives an antimessage for a message which it has already processed, it must perform a roll back to the antimessage time. However, if the message associated with the antimessage has not yet been processed, then the message and the antimessage simply annihilate each other.

During initial ParaTran research, the computational model selected could fairly accurately be described as an extended Kung and Robinson model optimized for support of parallel programming languages. This approach was selected because it seemed to solve the problem of apparent sequentialization while incurring less overhead than a Time Warp style model. However, we have found that the overhead is pathological in the sense that it is not sufficiently parallelizable to yield time efficient results on multiprocessors with a large number of computational units. As a result, we are currently working on modifying the ParaTran model of computation to show much greater similarity to the Time Warp model. In order to give the reader greater historical perspective and a better overall understanding of the issues involved in ParaTran, we will first present the initial Kung and Robinson based model of computation and will then discuss the similarities and differences between this and the Time Warp style model currently being developed.

The analogy between the original ParaTran model and the Kung and Robinson system can be best understood by viewing a Lisp program as a series of operations to be performed on a virtual database composed of the heap plus variable bindings (global store) utilized in executing the code. A program can be partitioned into a series of pieces, or tasks in the traditional programming language terminology, which are really just transactions against the virtual database. If the bodies of functions are subdivided into a number of transactions which themselves call other functions, then a nested transaction model results. (The importance of this nesting will become more obvious during later discussions of the details of the decomposition of programs into transactions.)

Under the original ParaTran model, the execution of tasks was divided into the same three phases as Kung and Robinson transactions. During the read phase, the code for a task was executed. All interrogations of variable bindings and data structures were recorded as queries to the database on a read list associated with that task. Attempts to modify variable bindings or data structures were shadowed by recording them on a write list, just as they would have been for a transaction system. Validation was again performed in the sequential order in which the tasks were supposed to appear to be executing. However, the validation and write mechanisms were slightly modified. During the write phase, the original ParaTran not only updated the required data items; but, it also broadcast the list of updated items. This allowed validation to be performed by comparing a transaction's read list to all of the broadcasts issued between the time its read phase was initiated and the time its validation began. These broadcasts were for those writes which were performed by transactions intended to run prior to the validating transaction on a uniprocessor machine, assuring correct sequential operation. This seemingly more complex approach was utilized because it tended to improve garbage collection efficiency and reduce the time spent repeating the read phase of transactions which failed to validate. (See [Katz86] for details.)

In order to better understand how the above model works, a brief example of how Lisp code might be partitioned into transactions will be presented. It should be noted that this description, presented in the Scheme dialect of Lisp, is for explanatory purposes only and that an actual partitioning for efficient execution would be significantly more complicated. Basically, Scheme programs are created utilizing only four constructs: function creation, function invocation, sequentialization, and conditionals. The sequentialization construct, sequence, is composed of a series of subexpressions. Due to the functional nature of the language, a sequence returns the value of its last subexpression. In this segmentation model, a transaction is built for the entire sequence, and a subtransaction is created for each subexpression of the sequence. A tree representation of the nested transactions generated for the sequence **(sequence a b c d)** can be found in figure 1. The root node of the tree represents the transaction for evaluating the entire sequence, and each of the leaves is a subtransaction for evaluating one of the subexpressions. The order in which the subexpressions which compose a sequence would have been executed on a uniprocessor can be reconstructed by reading the children in the transaction tree from left to right. In a more general tree, the sequential order in which the transactions are to appear to be executing can be uniquely regenerated by traversing the tree in preorder.

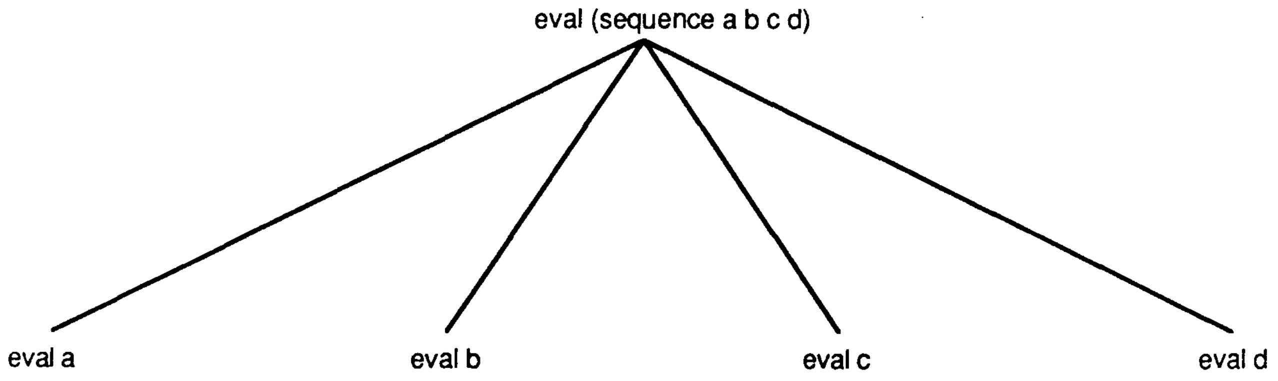


Figure 1. A transaction tree for **(sequence a b c d)**

The invocation of a function can be broken into three parts. First, the operator must be evaluated to determine what code should be used in the function application. This is necessary in the Lisp family of languages because the code for a function may actually be the result of executing some other piece of code. Next, each of the operands must be evaluated. These too could be the results of other computations. Once both the operator and operands are known, the actual function application can be performed. Consequently, a function call becomes a transaction for the evaluation of the function call which is composed of a series of subtransactions: one for the evaluation of the operator, one for the evaluation of each operand, and one for the application of the operator to the operands. An example of a transaction tree created for a simple function call of the form (append a b) can be found in figure 2. There is a very important piece of work which is done as part of a function application and adds to the complexity of the ParaTran model. This is that the function to be applied must be read before the function application can be performed. This read is one which must be recorded during the read phase of the application transaction. Clearly, if an incorrect binding between a function name and body is utilized in executing a function application, the transaction for that application must fail to validate, and instead be **aborted** and restarted with its read phase. In the event that the transaction for the function application is itself composed of multiple subtransactions, these subtransactions may no longer be meaningful once the parent has been aborted and should be **killed**. When a transaction is killed, it is permanently removed from the transaction tree and the system, instead of merely being restarted as would be the case for an abort. This is the reason that the nested nature of the transactions is critically important for executing Lisp.

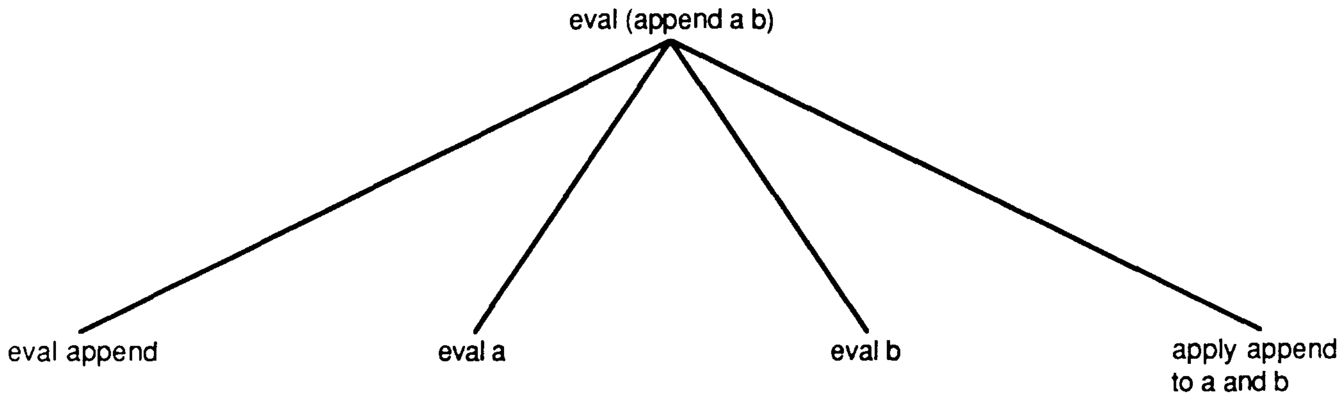


Figure 2. A transaction tree for (append a b)

While the handling of function applications seems to be very complicated, it yields great dividends in that conditionals can be implemented using the identical mechanism. Evaluation of a conditional involves two steps: evaluating the predicate and evaluating either the consequent or the alternative. In this model, a transaction will be created for every conditional and subtransactions for each of the two steps in evaluating the conditional. A transaction tree for a conditional of the form **(if pred a b)** can be found in figure 3. The subtransaction for the evaluation of the consequent or alternative can be handled just like an application transaction. This subtransaction has the same type of dependency on the predicate that an application has on its operator. If the value returned by a predicate changes due to the predicate transaction being aborted, and reexecuted, then the transaction for the alternative or the consequent needs to be aborted and any children it might have spawned must be killed.

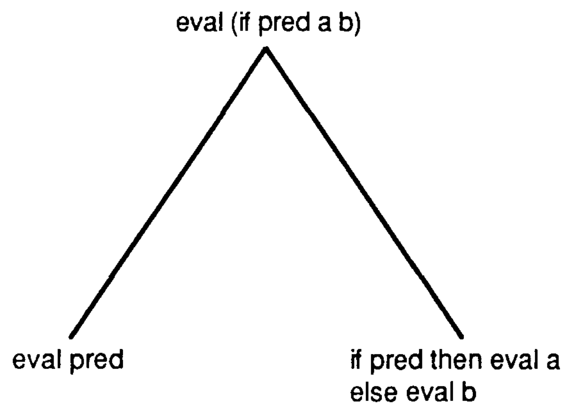


Figure 3. A transaction tree for (if pred a b)

A combination of the partitioning rules presented above is conveniently demonstrated by the recursive factorial program in figure 4. The transaction tree created for the function invocation **(fact 1)** can be found in figure 5. In order to maximize parallelism while minimizing the overhead due to aborts and kills, an embellishment was added to the original ParaTran model. Since many operations in a functional language only manipulate references to an object and never actually look at the object itself, some parallelism can be gained between function application and argument evaluation if tasks return placeholders into which they will store their results once they are calculated. **Futures** are a type of placeholder which have a special set of properties that make them desirable for this application [Halst84, Halst85]. A future is initially created without any value assigned to it and is said to be **undetermined.** If a task tries to refer to the value of an undetermined future, it becomes suspended and is placed on a queue of tasks waiting for a value to be assigned to the future. When a value is assigned to a future, the future becomes **determined.** All of the tasks which were waiting on the calculation of the future's value are now returned to a general work queue of tasks to be executed. The execution of these tasks is resumed from the point at which they attempted to access the value of the just determined future. An attempt to access the value of a determined future is completely transparent. The value is returned just as though the future object weren't present. This yields a controlled form of producer/consumer parallelism. By extending the semantics of futures slightly, they can be used to control the dependence of application tasks on operator evaluation tasks and of consequent and alternative tasks on predicate tasks. (See [Katz86] for details.)

```
(define (fact n)
  (if (zero?  n)
      1
      (* (fact (1-+ n)) n)))
```

Figure 4. A recursive factorial program

The reason we have chosen to move away from the initial ParaTran model to a model that shows more resemblance to Time Warp is that validation under the initial model was an inherently sequential processes. We originally felt that this could be overcome by making the validation phase of a transaction significantly faster (i.e. 2 to 3 orders of magnitude) than the read phase; but, we found that even this was not sufficient. The semantics of validation make it impossible to begin this process until the farthest leftmost leaf of the transaction tree has finished its read phase. By this time, it is often the case that so many transactions have already been created and begun processing that validation can never catch up sufficiently. We considered minor modifications to the scheme presented in which some parallel validation was possible; but, all of these suffered from either too much overhead, poor memory utilization/garbage collector problems, or both. Therefore, we are currently developing a ParaTran model which has more of a Time Warp flavor. Whereas the previous model was based on the synchronization of tasks or transactions, the new model depends on a more object based synchronization.

A fairly simple mapping between the Time Warp model of computation and symbolic programming languages can be made if one views the Lisp heap and the code of a program

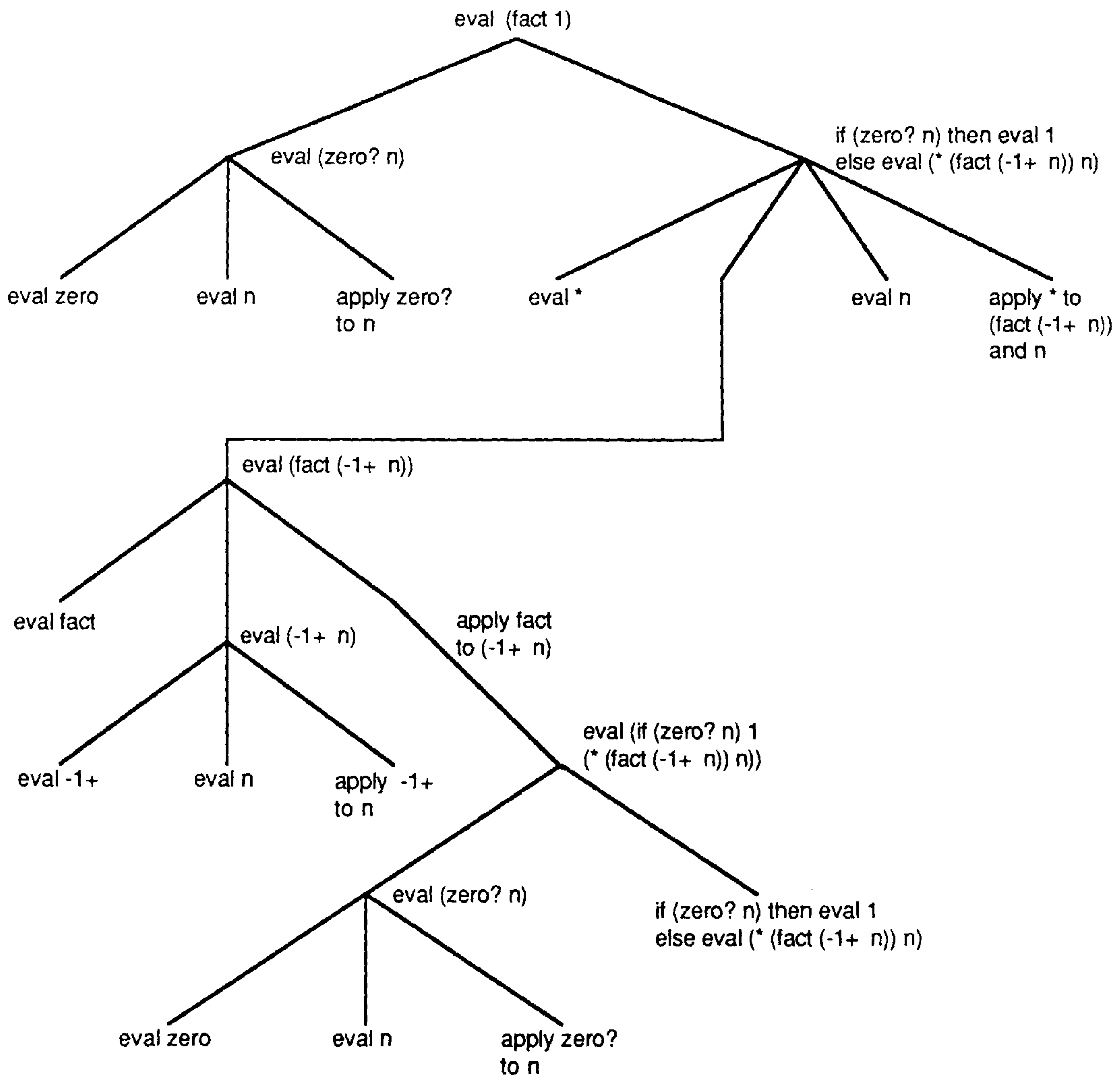


Figure 5. A transaction tree for (fact 1)

as being composed of objects which communicate via messages. Each memory location of the heap is effectively an object which can process two types of messages: **read messages** and **write messages**. These objects respond to read messages by sending a **value message** to the requester; and, to write messages, by modifying their internal state. Each transaction which composes a program is also essentially an object which can issue and respond to several types of messages. A transaction can issue read and write messages to a heap object, **create messages** to create another transaction object, and **create messages** to create a heap object. Transaction objects can also process value messages sent in response to read messages which they have issued. By analogy with Time Warp, it can be seen that an antimessage to a create message is actually a **kill message**, so the abort/kill based nested transaction model continues to be supported by the new ParaTran model.

In order to maintain transparency, it is important that the ParaTran system handle time stamp generation for the user, unlike Time Warp. Each transaction is issued a time stamp, by the ParaTran system, when it is created. These time stamps form a totally ordered set from which the a priori sequential order in which the transactions would have executed on a uniprocessor can be reproduced.

So far, the new ParaTran model has been presented in a somewhat abstract and idealized fashion. The actual implementation has been designed to take advantage of idiosyncrasies of the programming language domain. Heap objects will be represented by a read and a write list. The write list will contain one entry for each time the object receives a write message. These entries will contain the value written, a time stamp for the virtual time at which it was written, and a pointer to the transaction object which sent the write message. Also, a special write entry will be created when a heap object is created and instantiated with its initial value. This entry will have no time stamp, identifying it as the initial value. The read list will contain a single entry for each time a heap object receives a read message. These entries will be composed of a read time and a pointer to the transaction object which issued the read message. (See figure 6 for a diagram of a heap object in box-and-pointer notation [Abels84]). When a heap object receives a read message, it will use the time stamp of the read message in searching its write list to find the lastest write, in time stamp order, prior to the read time. It will then send a value message back to the requesting transaction object with the value written at that time and record the read in the read list of the heap object. Unlike Time Warp, the fact that a heap object receives a read message with a time stamp that is earlier in the temporal ordering than an already processed read or write message does not necessitate a roll back of the heap object under ParaTran. However, receipt of a write message by a heap object may lead to roll back. When a write message is received, the heap object first adds an entry to its write list. It then searches its read list for any reads with time stamps between the time at which the new write is to take place and the time of the earliest write in the temporal ordering which follows the new write. It is these reads which potentially returned an incorrect value. Therefore, the transactions which issued these reads are sent **antivalue messages** causing them to roll back to the time at which they read the erroneous value.

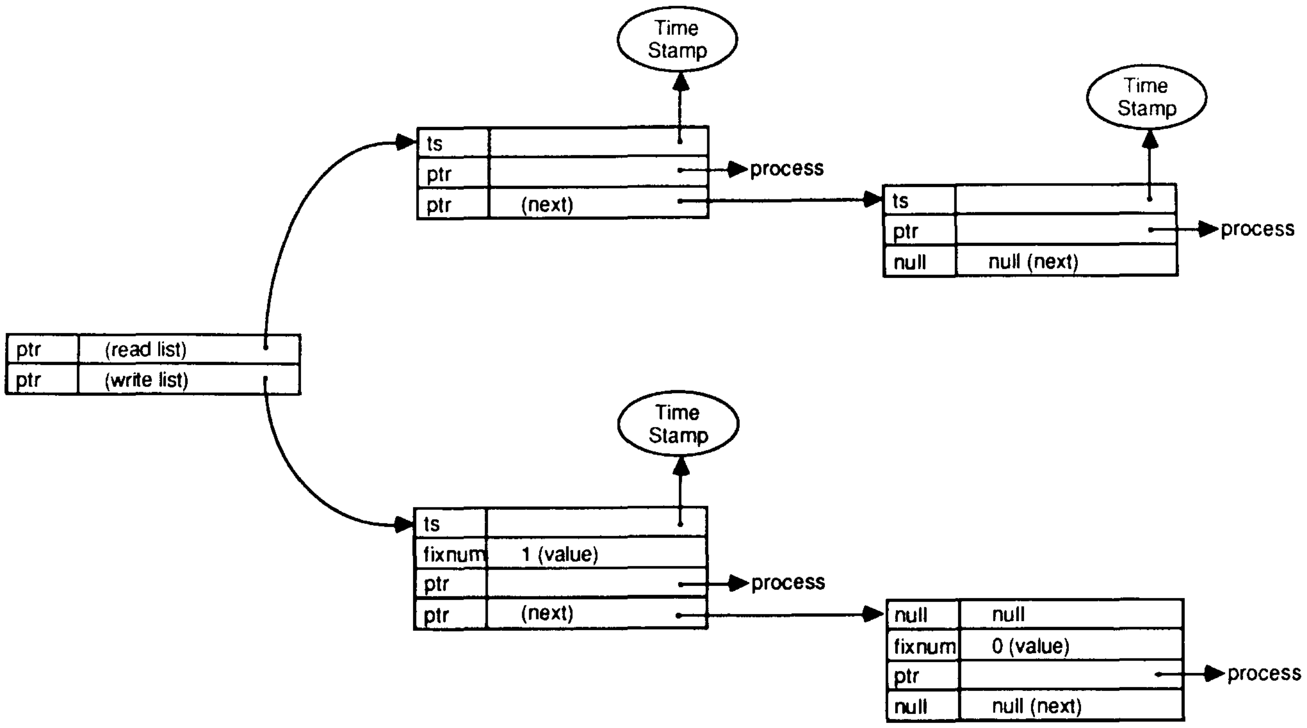


Figure 6. The box-and-pointer representation for a slot under ParaTran

The roll back of a transaction object can cause the issuance of three types of antimessages: **antiread, antiwrite, and anticreate messages**. Antiread messages merely cause the appropriate entry to be removed from the read list of the appropriate heap object. Antiwrite messages cause a similar removal of an entry from the write list of a heap object. However, this action might invalidate some previously processed read messages. Therefore, the read list of the heap object must be searched for any reads processed at virtual times between the time of the retracted write message and the time of the next write message, in time stamp order, following the retracted write. Any reads in this class are invalidated by sending antivalue messages to the transaction objects which issued the reads. Anticreate messages are basically kill messages to the children of the transaction object which is being aborted. They are processed by rolling back the transaction object which receives them to its inception and then deleting the object from the system. It should be noted that since the issuance of a message or an antimessage can only cause rollback to a virtual time which is greater than or equal to the time stamp of the message, the system presented is guaranteed to be deadlock free. Furthermore, any Lisp program which terminates on a sequential machine is guaranteed to also terminate under ParaTran.

Having presented the state-of-the-art in optimistic parallelism and transaction systems and our current understanding of ParaTran, what remains is to outline the areas in which the most intensive research remains to be done. Unlike the domain of discrete-event simulations where Time Warp users are able to specify either absolutely or programmatically the virtual time at which each event is intended to take place, the Lisp programmer often does not know this information and should not have to be concerned with it. In Lisp any piece of code or transaction may create any number of subtransactions which may in turn create a limitless number of subsubtransactions, etc. The unboundedness of the width and breadth of transaction nesting means that any interval of time stamp space must be partitionable into an infinite number of subintervals. We are currently considering several time stamp representations which posses this property. Of paramount importance in this effort is that any time stamp representation must allow for fast, efficient time stamp generation and comparison. In all likelyhood, a remapping of time stamps at garbage collection time may be required to meet these goals.

The partitioning of a program into transactions is another problem whose analog is much easier in the discrete-event simulation domain. In Time Warp the user explicitly defines all of the objects and the ways in which they should respond to messages of various types. Under ParaTran, we intend for the compiler front-end to do much of this work. Since efficiency is nearly always of great importance to the user of a multiprocessor, the programmer will be able to help the compiler perform this process more efficiently through the use of pragmas. Although it is our hope that the compiler will be able to perform this job well enough that pragmas will not be needed to any great extent, only very limited success in this area has taken place within the computer science community over the last 10 years. It is for this reason that we feel it is important to include support for user pragmas in the design of ParaTran. However, it should be pointed out that the problem of transaction partitioning within ParaTran is much easier than the process partitioning problem faced by the programmer using an explicit parallelism language. This is because ParaTran partitioning only affects efficiency; whereas, partitioning in explicit parallelism languages affects both efficiency and correctness of operation.

Efficient garbage collection is another area in which ParaTran will need to perform a task that is left largely to the user by Time Warp. In discrete-event simulations, the creation and deletion of objects is specified explicitly by the programmer; but, in ParaTran this must be handled implicitly by the system. In particular, the lifetimes of heap objects, and therefore the transaction objects which manipulate them, are inherently dynamic. Consequently, a sophisticated garbage collector is being designed which will delete transactions which can no longer be aborted and heap objects which can no longer be referenced. This requires that the garbage collector utilize extensive knowledge of time stamps and perform operations more complex than the **global virtual time** processing done by Time Warp [Jeffe85].

A concept which does not arise in discrete-event simulations, but is very important in programming languages, is that of tail recursion. A reduction is a situation in which a function calls another function in such a manner that the caller will return whatever result is returned by the callee. A recursive implementation of a function to find the last element of a list (see figure 7) is a classic example of reduction. Each invocation of the function reduces into the next, until the last element is found. A tail recursive interpreter takes advantage of its knowledge of reduction in order to limit the amount of stack space required to execute a program. When a reduction is performed, the only portion of the stack frame of the caller which is necessary following the call is the pointer to the piece of code to which control should return following the completion of the execution of the callee. None of the values of the local variables or arguments to the caller are of any importance. Therefore, the stack frame of the caller can be eliminated if the return pointer of the callee is made to be the return pointer of the caller. This approach not only saves stack space, but also means that returning from a deeply nested set of reductions costs constant time rather than time proportional to the depth of the recursion. How to effectively support tail recursion in ParaTran is one of the most difficult issues that we are currently investigating.

A final area of exploration for the ParaTran project is how to perform efficient scheduling of transactions on the available processing resources. This is an area of research which

```
(define (last list)
  (if (null? (cdr list))       ;If the end of the list has been reached:
      (car list)               ;Return the last element of the list
      (last (cdr list))))      ;Else, search the rest of the list
```

Figure 7. A tail recursive program to find the last element of a list

we share with most of the rest of the parallel processing community. We suspect that the optimal scheduling decisions for ParaTran may differ significantly from those for Time Warp, and many explicit parallelism systems, because of differences in the problem domains. In particular, all of the transactions to be processed by ParaTran have an a priori sequential order in which they would have run on a sequential computer. Such information is not available to many other systems where the situation being simulated is usually inherently nonsequential.

In conclusion, we believe that ParaTran offers a promising and novel approach to solving the symbolic parallel processing problem. It is both based on and contributes to the state-of-the-art in the areas of transaction systems and optimistic concurrency. It is our expectation that once again ParaTran will demonstrate the power of combining innovative and creative ideas in several related fields.

PROJECT OVERVIEW

We view ParaTran development as being composed of the following six tasks:

- design of the computational model
- implementation of a transaction partitioner
- building of a functional simulator
- programming of a performance simulator
- creation of an actual parallel version
- generation of a compiler for the parallel version

The first step once one has designed a model is always to verify the efficacy of that model. We intend to perform this task through the building of a functional simulator and transaction partitioner for ParaTran. The transaction partitioner will act as a front-end for either a compiler or interpreter and will partition a user's program into transactions. This process will be performed based on compile time flow analysis and user supplied pragmas. The output of the partitioner will then be fed to a time-slice based functional simulator which will be used both to determine if the ParaTran model has any functional deficiencies and to enable us to refine the model. The functional simulator is not designed to give accurate performance measurements; but, instead, to help guide us in model development. This less stringent set of requirements means that the functional simulator can be implemented in Lisp, greatly reducing simulator development time and lessening the effort required to make modifications to the functional simulator.

Once the ParaTran model has been brought to a fairly mature state through the use of the functional simulator, we will be ready to build a performance simulator. This will be done by adding some constructs which are idiosyncratic to ParaTran to an existing Lisp system and by modifying the implementation of some of the existing language primitives. We have selected the MIT CScheme dialect of Lisp for use in this and later stages of ParaTran development for a number of reasons.

- MIT CScheme is in the public domain so the source code is available.
- The Lisp interpreter is written in C so it is easy to make additions to the language and to port it to different machines and operating systems. (MIT CScheme is currently supported on over half a dozen machine/operating system pairs.)

- MIT Cscheme is implemented as a small core of primitive actions on which the higher level constructs are built. This implementation methodology makes customization and modification of the system much easier than working with a large, unwieldy language.
- MIT CScheme contains many useful features for the development of ParaTran which are not found in other Lisps (e.g. first class, reusable continuations).
- CScheme was the basis of the DARPA sponsored Butterfly Commonlisp Project at BBN. As a result of this effort, MIT CScheme has several other advantages to offer us.
    - BBN developed a Commonlisp compatibility package for MIT Cscheme as part of their work so ParaTran will effectively be able to run both Scheme and Commonlisp.
    - A number of parallelism primitives were added to CScheme in order to generate Butterfly Commonlisp. Many of these such as futures, weak pointers, and semaphores will be used by ParaTran.
    - A parallel garbage collector was developed for Cscheme on the Butterfly.
    - The fact that MIT CScheme has already been ported to the Butterfly means that the only portions of ParaTran which we actually have to build for a version on a Butterfly are those which are idiosyncratic to our system.

The performance simulator will be used both to help in further refining the ParaTran model and to test the scalability of ParaTran to machines having a larger number of processors that those currently readily available. However, the only accurate test of any parallel system is an actual multiprocessor implementation. We will therefore be creating such a version of ParaTran for a BBN Butterfly. Our previously mentioned decision to implement the performance simulator by adding primitives to CScheme means that the parallel version can, to a large extent, be built just by porting these constructs to the multiprocessor. A Butterfly was selected as the target machine for two primary reasons:

- As stated above, MIT CScheme has already been ported to the Butterfly. This means that a significant implementation effort has already been performed for us.
- The Butterfly's support of shared memory makes the development of a Lisp system significantly easier. We feel that to undertake the complications inherent in a distributed memory Lisp as well as those associated with the development of a sophisticated system like ParaTran would be unwise at this time.

However, a port to other parallel machines is certainly possible. We believe that ParaTran could easily be supported both on a hypercube using simulated shared memory and on the Ultramax being developed by Encore for DARPA under the Multiprocessor System Architecture R&D Project (contract #N00039-84-R-0605(Q)).

The final stage in ParaTran development will be the creation of a compiled version of the system. We intend to perform this stage of the research by augmenting and extending the CScheme compiler which is currently being developed at MIT. This compiler is also, in all likelyhood, going to form the basis of the MultiScheme compiler development project. This should again lead us into a good symbiotic relationship with the MultiScheme research community.

REFERENCES

[Abels84] Abelson, H., and G. Sussman, *Structure and Interpretation of Computer Programs*, M.I.T. Press, Cambridge, Mass., 1984.

[Abels85] Abelson, H., *et al.*, "The Revised Revised Report on Scheme or An Uncommon Lisp," M.I.T. Artificial Intelligence Laboratory Memo 848, Cambridge, Mass., Aug. 1985

[Gabri84] Gabriel, R. P. and J. McCarthy, "Queue-based Multiprocessing Lisp," *ACM Symposium on LISP and Functional Programming*, Austin, TX, August 1984, pp. 25-43.

[Halst84] Halstead, R., "Implementation of Multilisp: Lisp on a Multiprocessor," *ACM Symposium on LISP and Functional Programming*, Austin, TX, August 1984, pp. 293-298.

[Halst85] Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, Oct. 1985, pp. 501-538.

[Jeffe85] Jefferson, David R., "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 404-425.

[Katz86] Katz, Morris J., *ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme*, MS thesis, Massachusetts Institute of Technology, June 1986.

[Kuck80] Kuck, D. J., R. H. Kuhn, B. Leasure, and M. Wolfe, "Analysis and Transformation of Programs for Parallel Computation," *Proceedings of the Forth International Computer Software and Applications Conference*, Oct. 1980.

[Kung81] Kung, H. T. and Robinson, John T., "On optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981, pp. 213-226.

[Lisko83] Liskov, Barbara and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 381-404.

[Moss81] Moss, J. Eliot B., "Nested Transactions: An Approach to Reliable Distributed Computing," MIT/LCS Technical Report #TR-260, April 1981.

[Reed79] Reed, David P., "Naming and Synchronizing in a Decentralized Computer System," MIT/LCS Technical Report #TR-205, Sept. 1979.

[Scott86] Scott, Curtis, "Butterfly Lisp Reference Manual," BBN Laboratories, Cambridge, Mass., April 21, 1986.

[Stein86] Steinberg, Seth, *et. el.*, "Butterfly Lisp System," *Proceedings of AAAI*, 1986.