NASA SUPPORT

IN-62-CR
197493
157P

Software Coherence in Multiprocessor Memory Systems

William Joseph Bolosky

Technical Report 456
May 1993

UNIVERSITY OF ROCHESTER COMPUTER SCIENCE

Software Coherence in Multiprocessor Memory Systems

by

William Joseph Bolosky

Submitted in Partial Fulfillment

of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

Supervised by Professor Michael L. Scott

Department of Computer Science

College of Arts and Science

University of Rochester

Rochester, New York

1993

To R. R. Camp

Curriculum Vitae

William J. Bolosky was born in [redacted] on [redacted]. He attended California State College in California, Pennsylvania from 1977 through 1983. He completed a Bachelor's degree with Univerity Honors in Mathematics at Carnegie-Mellon University in 1986. After working as a research staff member with Carnegie-Mellon's Mach project, he began graduate studies at the University of Rochester in the fall of 1987, studying Computer Science under Professor Michael L. Scott. In 1989, he received a Masters of Science in Computer Science from the University of Rochester. In 1992, he accepted a position as a Researcher with the Microsoft corporation in Redmond, WA.

He received a Sproull Fellowship for graduate studies at the University of Rochester in 1987, and a DARPA/NASA Fellowship in Parallel Processing in 1991.

Acknowledgments

While my name is the only one listed as the author of this document, it is hardly the case that all the work described herein is mine alone. Rather, the bulk of this dissertation is derived from work published jointly with others. Chief among them is my advisor, Michael Scott; without his help none of this could have happened. During the ACE project, Bob Fitzgerald was a constant source of inspiration and thought-provoking criticism. Rob Fowler and Alan Cox were very helpful in refining the work. Tom LeBlanc has been helpful throughout my tenure at Rochester as a sounding board for my ideas, and Charles Merriam was there when he was needed. Alexander Brinkman provided an extremely helpful, very thorough reading of the dissertation prior to my defense. Cesary Dubnicki and Jack Veenstra provided insight into my traces and the applications from which they were generated, as well as the general topic of shared memory multiprocessor memory systems. Prior to my arrival at Rochester (and subsequent to my departure), Rick Rashid served both as a mentor and inspiration to me.

The work was supported by a University of Rochester Sproull Fellowship, two IBM Summer Internships and a DARPA/NASA Fellowship for Parallel Processing. Furthermore, the ACE hardware used for much of this work was generously lent to the University of Rochester by IBM, through the good graces of Armando Garcia, Bob Fitzgerald and Colin Harrison.

Abstract

Processors are becoming faster and multiprocessor memory interconnection systems are not keeping up. Therefore, it is necessary to have threads and the memory they access as near one another as possible. Typically, this involves putting memory or caches with the processors, which gives rise to the problem of coherence: if one processor writes an address, any other processor reading that address must see the new value. This coherence can be maintained by the hardware or with software intervention. Systems of both types have been built in the past; the hardware-based systems tended to outperform the software ones. However, the ratio of processor to interconnect speed is now so high that the extra overhead of the software systems may no longer be significant. This dissertation explores this issue both by implementing a software maintained system and by introducing and using the technique of *offline optimal analysis* of memory reference traces. It finds that in properly built systems, software maintained coherence can perform comparably to or even better than hardware maintained coherence. The architectural features necessary for efficient software coherence to be profitable include a small page size, a fast trap mechanism, and the ability to execute instructions while remote memory references are outstanding.

Table of Contents

List of Tables

List of Figures

1 Introduction

Shared memory multiprocessors have become increasingly popular in recent years. The straightforward parallel programming style they present is attractive to users. To supply the necessary low-latency, high-bandwidth memory to the processors, architects have devised several schemes in which data is replicated to various types of memories which are local to the processors. An effect of this approach is that the system must make sure that changes in data made by one processor are visible to other processors. This is called the "coherence problem."

There have been several different approaches to the coherence problem in the past. They fall into two major categories, depending on whether they are implemented in hardware or software. Traditional hardware solutions use caches in which coherence is maintained either by requiring that data not be replicated at the time it is written (called "Write Invalidate [4, 31, 69, 60]"), or by broadcasting all writes to replicated lines to all holders of those lines (called "Write Update [76, 95]"). Software solutions to the coherence problem use a special operating system kernel that migrates and replicates data [14, 39, 58, 64], much as a cache controller would in a hardware implementation. A third method of solving the coherence problem is remote reference, in which single words are read or written by one processor directly in another processor's memory, usually at a significant time penalty over local memory [14, 15, 28, 50, 82]. Remote reference is more commonly used in software-coherent systems than in coherently cached ones.

Most new designs use hardware coherent caches. Hardware is able to process coherence operations with less overhead than software, and simulation studies show that the hardware based systems perform better overall. However, this dissertation argues that software coherent machines, if properly designed, can in fact perform competitively with hardware coherent ones, and that the added flexibility and reduced cost of construction of software based systems make them a viable alternative to hardware coherent machines. The two main reasons are: 1) processors have become so fast relative to interprocessor interconnections that the difference in time to initiate a coherence operation between software and hardware coherent machines is swamped by the time for the coherence operation to complete; and 2) the primary reason that hardware coherent machines do better than their software-controlled cousins is that a hardware cache line is typically much smaller than a software page, and this results in a great reduction in false sharing,[1] and consequently better performance. There is no *a priori* reason that pages need to be so big,[2] and so no reason that the hardware/software performance difference needs to be as large as it is.

Therefore, my thesis statement is: "Properly designed software-coherent shared-memory multiprocessors can perform competitively with coherently cached machines."

1.1 Data Locality and the Coherence Problem

Communication between processors is the fundamental problem in multiprocessor design: it is communication that limits the ability to build arbitrarily large parallel machines. As the need for communication lessens, the number of processors that can profitably be employed grows. If no communication at all were necessary, then the only limits on parallelism would be algorithmic and monetary. Unfortunately, in practice this is not the case, and attention must be paid to inter-processor communication.

Think about the task of designing a traditional, tightly coupled multiprocessor in terms of the communication requirements. To be able to add more utile processors, it is necessary either to increase the "amount" of communication possible between the processors, or to reduce the amount that is needed. Increasing the capacity of the hardware is, and will continue to be, a profitable activity, but how to do it is beyond the scope of this work. Reducing the amount of communication needed can be done in two basic ways: finding algorithms that solve the desired problems with less communication, or by building machines that exploit locality in programs' data usage. Changing parallel algorithms to reduce the amount of communication needed is also a profitable avenue of exploration, and is also beyond the scope of this work.

After excluding modifying interconnections and algorithms, exploitation of data locality in programs remains as a method to reduce communication, and so increase the utility and size of multiprocessors. The idea is that some memory

[1]False sharing is sharing that happens solely because of colocation of data within a coherence unit; it is the topic of Chapter 7.

[2]The reason that pages are large in uniprocessor implementations is to reduce number of per-page operations such as faults, page table maintenance, allocation of per-page virtual memory system data structures and so forth. While I propose smaller hardware pages in order to reduce false sharing, I believe that a virtual memory system on such a machine should group hardware pages together into larger software pages for all purposes other than coherence.

will probably be used exclusively by some processors, some will be used by many processors, but not change often, some will be used for a time by one processor and then later by another. Using variations on the theme of caching, it is possible to reduce the amount of communication that is needed for data that fall into these categories. Data that are used by just one processor may be stored at that processor; references to the data needn't affect any other processor in the system. Data that are being read by many processors, but modified by none, may be replicated to each of the processors using the data, and as long as they are not modified there need be no non-local operations. Migratory data can be moved from processor to processor when their user changes. Data that are shared at a fine grain are problematic, and will be discussed later.

Unfortunately for system builders, application writers do not specify the reference behavior of each region of memory of their programs. In fact, it is a goal of shared memory multiprocessor design to present the illusion of a single shared memory to the application programmer, even though the machine is more complex. Therefore, operating systems (including compilers) must determine the reference pattern for a particular region of memory by inspecting the references made to it as the program executes. A requirement of this inspection is that it enforce coherence: when a processor reads a datum, the value that it gets is the latest written into the particular address by any processor.[3] Since it is possible that this other processor has its own copy of the data, it is necessary for the system to communicate that an update has happened before the subsequent read occurs.

1.1.1 Methods of Implementing Coherence

There are three primary techniques for maintaining coherence in a multiprocessor memory system, any of which can be implemented in either software or in hardware. They are write-update, write-invalidate and remote reference.

In write-update the writer of a word sends the value that is written to other holders of the data, who then update their copies. The sending of data may be accomplished by broadcast, multicast or sequentially sent messages. Write-update performs well in systems that have very little writable memory, or in systems such as cached bus-based multiprocessors, in which the cost of sending frequent updates is very low. Software implementations of write-update typically either exploit weak coherence [19], or are intended to be used only for a small set of data structures that the programmer has determined will profit from write-update [38].

[3]There are systems that do not satisfy this constraint, but rather require the programmer to either annotate programs in some way, or to understand the weaker coherence constraint [2, 19, 70, 99].

Remote reference is like write-update in that it handles individual memory references through the interconnection network. However, unlike write-update a given line or page cannot be replicated. Rather, all references to the memory are directed to a single location where the data are kept. References (both reads and writes) made by the processor holding the data are local; those by any other processor go through the interconnection. Remote reference is most useful when data are used heavily by only one processor, and only rarely by all others.

Write-invalidate is the most popular technique for maintaining coherence. It operates by ensuring that a writer of data has the only copy of that data at the time of the write. That is, any other copies at other processors are invalidated before the write proceeds. Write-invalidate is a very popular technique, and is used in both hardware and software based systems. It handles migratory and read-only data properly.

These methods may be combined to produce hybrids. Veenstra and Fowler [98] compare strict write-invalidate, strict write-update and a few different hybrids of the two and conclude that the hybrid strategies perform better than either pure version. NUMA (Non-Uniform Memory Access) machines, which are the subject of much of the work described herein, are a hybrid of write-invalidate and remote-reference. These types of hybrids are faced with the additional problem of determining when to use each of the methods: should a block be replicated, moved or remotely referenced? When should a write be broadcast (*i.e.*, use write-update) and when should remote copies be invalidated? Policies that answer these questions in NUMA systems are discussed in Chapter 5 and in various others' work [39, 58, 57, 63, 64, 65].

An advantage that software implementations of coherence has over hardware implementations is that they may implement more complicated protocols, and may vary their protocol more easily. For example, PLATINUM uses a timer-based exponential backoff algorithm for page thawing [38]. This would be harder to do in hardware.

1.2 Outline of the Argument Supporting the Thesis

The simulation studies in Chapter 6 demonstrate directly that software coherent machines perform comparably to hardware coherent machines. The bulk of the dissertation describes the simulation methods, justifies them and presents and analyzes their results.

Any trace-driven simulation study is only as valid as the applications used to drive the simulation, and the accuracy of the traces of these applications. Chapter 2 describes the applications and the tracer used to collect the memory references from them. Chapter 3 describes an implementation of a kernel on a NUMA system. It shows the extent to which such a system may be evaluated, and points out the need for a different approach in order to gain a more detailed understanding. Section 4.3 describes a series of experiments designed to determine the effect of errors that might have been made during the tracing process, and concludes that they are likely of sufficiently small magnitude that they are inconsequential relative to the size of the effects seen in the main experiments.

When evaluating a simulation study, the important questions are: Does the simulation model important characteristics of the system? Is the model accurate? What are the sources of error in the simulation, and to what magnitude are they present? Are the applications used to drive the simulation representative?

Chapter 4 describes a formal model of machines, policies, programs, and their performance. An important characteristic of machines that have remote reference and block migration is that it is necessary to decide when to use one and when to use the other. This is the function of the policy component of the model. The behavior of the policy can greatly influence the overall performance of the system. Therefore, the concept of an optimal policy is introduced and is used to eliminate the bias inherent in any non-optimal policy. Briefly, the optimal policy uses future knowledge in order to always make the appropriate remote reference/page move choices, thereby minimizing cost for the particular application and machine model being simulated. This optimal policy is also presented in chapter 4.

For the results of this optimal analysis to be interesting, it is necessary that on-line, implementable policies have performance that is reasonably close to that of the optimal policy. Demonstrating that on-line policies can indeed approach optimality, together with results about just what policies are appropriate for what architectures, is the topic of Chapter 5.

Chapter 6 applies the simulation method to a set of machine models based on components of speed comparable to that available in modern product hardware. The machines studied differ in their use of software vs. hardware coherence, in their support of remote reference and in the page or cache line sizes used. The result of comparing their relative performance is that software coherent machines (with sufficiently small pages) perform comparably to hardware coherent machines, thus confirming the thesis statement.

While the study in Chapter 6 shows that NUMA machines with sufficiently small pages perform comparably to coherently cached machines, it offers little insight as to why this is the case. Chapter 7 offers and explores the hypothesis that false sharing is the reason that reduced page size is so beneficial. While I was unable to come up with a satisfactory formal definition of false sharing that is known to be computationally tractable, Chapter 7 offers evidence that false sharing is present in some applications; that evidence is also strongly suggestive that false sharing is the major contributor to the performance difference between

NUMA and hardware coherent machines. Chapter 8 presents a summary and conclusions, and mentions directions for extending the work.

1.3 Related Work

The question of software-implemented coherence touches upon several different related areas, including multiprocessor architecture (specifically multiprocessor cache and interconnection design), multiprocessor operating system design and implementation, tracing and simulation methodology, and processor architecture.

1.3.1 Coherent Caching

There are several different ways in which one may implement coherent caches. The simplest is a *write-through snoopy* cache. That is, a cache in which every write is transmitted through the local cache to a broadcast bus connecting main memory and all other processors. All processors listen ("snoop") for writes by other processors that would affect a word in the local cache. When such a write is detected, the word is either updated to have the new data (in write-update schemes) or the line containing the word is evicted (in write-invalidate schemes). The Sequent Balance [96] uses write-through snoopy caching.

The problem with straightforward write-through snoopy schemes is that all writes, regardless of whether they are to shared data, generate broadcast traffic on the interconnection. Censier [31] observed that many of these writes could be local, and suggested associating a state bit with each cache line that indicated whether this processor had the only copy of the line; if it did, then the write went into only the local cache. If another processor wanted a line that was dirty and held by another, it forced a coherence operation. Censier's scheme still requires a broadcast medium, but eliminates much of the unnecessary traffic. It is a *write-back* snoopy technique. Goodman [54] proposed the write-once scheme, a cross between write-through and write-back. It worked by writing-through on the first write by a processor to a cache line, but if a second write occurred before a coherence operation for that line, it was changed to the dirty state, and eventually written-back. The idea of write-once was that fine-grain shared data would not "ping-pong," while most of the reduction of traffic of write-back would be retained.

Switching from write-through to write-back (or write-once) greatly reduces the amount of bandwidth necessary for a system of a given size. However, in systems with a large number of processors, implementing broadcast is difficult or impossible [14, 82, 102]. Typically, these systems use an interconnection with some sort of hierarchical structure, wherein messages are routed point-to-point. To implement coherence on such systems, it is necessary to drop not only the idea of write-through, but also that of snooping. Instead, a "cache directory" is used. The directory contains information about which processors are using which data, and what state they are in. This allows notice of coherence operations to be sent only to those processors needing to see them. There are various schemes for how to build and distribute cache directories and for the particular protocols used with them [10, 60, 69, 76, 86, 95, 99], but they are beyond the scope of this section. Surveys and overviews of multiprocessor caching [11, 91] are available for those desiring more information.

1.3.2 NUMA Systems

The acronym "NUMA" stands for Non-Uniform Memory Access. It typically refers to machines that have per-processor memories that can be referenced by all processors. In addition, NUMAs may have memory that is not associated with any particular processor, as discussed in [79]. In NUMA machines, it is the responsibility of software (either at the operating system, runtime or application level) to determine the (potentially dynamically changing) location(s) of data within the system's memory. That is, NUMA machines are software coherent systems that have remote reference. This section describes others' work on NUMA software and analytic modeling.

One of the earliest attempts at NUMA memory management was that of Holliday [58]. His implementation was also on a Butterfly multiprocessor, but differed from later efforts in that it did not use fault behavior to directly drive page location decisions. Rather, he implemented a daemon that ran periodically and determined recent reference behavior for pages by examining the memory manager's referenced and used bits, and then moved and replicated pages based on this daemon's findings. In a later paper [57] he concluded that it was very difficult to exploit migratory behavior in NUMA systems, and that simply establishing a good static placement would result in performance as good as that which could be achieved by handling migratory pages. Furthermore, he asserted that it is better to have the programmer explicitly specify the appropriate location for memory, rather than having the system find it, and consequently that NUMA systems were a bad idea. However, it is unclear that his conclusions apply to fault-driven NUMA systems, and in fact they conflict directly with the conclusions of Cox and Fowler [38, 39, 48], and LaRowe *et al.* [63, 64, 65].

In [23], Sleator and Black describe an algorithm for competitive NUMA management. That is, they describe a placement policy whose worst case behavior is as close to optimal as possible for any on-line policy. They describe algorithms that come within a factor of 2 in the case of read-only pages and within a factor of 3 for writable pages. However, in the writable case they require hardware that may make a remote transaction on every reference to otherwise local memory, which could cost a large amount of execution time. Furthermore, the DUnX results (and those in [27], which are also presented in Chapter 5) show that when dealing with real applications it is almost always possible to do much better than the competitive best-worst-case limits.

Black, Gupta and Weber use competitive NUMA policies in a simulation study described in [22]. It is difficult to judge the success of their methods because they only present their results as compared against a random placement of both code and data (in which read-only code pages are not replicated, and so instruction fetches have to proceed through the memory interconnection), and report tremendous speedups.

Modern NUMA operating systems make periodic page placement decisions based both on faults made by processors accessing a page that is not mapped and also on asynchronous events, typically timer interrupts (and pageout requests). The main question dealt with in the NUMA literature is that of the NUMA policy: when one of these events occurs, what action (if any) should be taken? Duke University's DUnX system was created as an experimental testbed for comparing these policies. It is an operating system that runs on the BBN Butterfly family of NUMA multiprocessors and in which the NUMA policy is very easily replaced. The authors wrote a large number of different NUMA policies and compared their performance over a suite of applications. Their results are described in [63, 64, 65]. Their principal conclusion is that most reasonable policies have comparable performance for most applications, and that therefore further study of NUMA policies is essentially pointless. Their explanation of this result is that most of the data used by a NUMA program is either private or only-read,[4] and any reasonable policy will place such data correctly. However, on machines with greater remote access penalties, such as those considered in Chapter 6, even a small amount of misplaced data can have very large performance effects, and so policy design is still important on such machines.

Cox and Fowler's PLATINUM system [38, 39, 48] is also an implementation of a NUMA kernel on the Butterfly. It differs from DUnX in that it was not created as a testbed for the comparison of a large number of NUMA policies, but rather as system with a small number of well tuned-policies for running all applications. The main PLATINUM NUMA policy operates by replicating pages when they are read, and migrating them when written by other than their sole owner. If a page is invalidated twice within a certain period of time, it is frozen in place and all references made from processors other than the one that happens to wind up with the only copy of the page are made remotely. This freeze decision is periodically reconsidered: in older versions of PLATINUM, all frozen pages were thawed by

[4] "Read-only" and "only-read" have different meanings. For a page to be read-only it must be so specified to the system at the time the page is allocated. On the other hand, to be only-read, a page simply must not be written; there is no requirement that the system be informed that this will be the case. Therefore, all read-only pages are only-read, but not necessarily the reverse.

a defrost daemon that ran at fixed intervals; the newer versions reduce "ping-ponging" of fine-grained shared pages by using an exponential backoff scheme for defrosting. A software implemented write-update scheme was also tried, and was found to work well for data structures that were mostly read but occasionally centrally updated.

1.3.3 Distributed Shared Memory

The idea of presenting a shared memory model on a machine or group of machines whose hardware does not support coherent caches or remote reference (and that, in many cases, was not designed as a multiprocessor system at all, but rather as a collection of ostensibly independent networked workstations) is called distributed shared memory. The seminal work in this area was done by Kai Li [72]. Since then, there have been several implementations.

IVY [71] was Li's first implementation, and used a network of Apollo [9] workstations running Apollo's Domain system. It was a prototype implementation, and supported process migration as well as shared memory. He later implemented DSM on an Intel iPSC/2 hypercube [73]. Forin, *et al.* [46] implemented a distributed Shared Memory Server on top of Mach's external paging system [103] with a straightforward write-invalidate protocol. This work was interesting because it was implemented without modifying the kernel, simply by writing an appropriate user-level external pager. Fleisch and Popek's Mirage system [45] is implemented using a modified Unix System V kernel [12]. Mirage's write-invalidate scheme is interesting in that it guarantees that when a writable copy of a page is created, that copy will not be invalidated for a certain amount of time. Any other processors that request that page during that time interval are suspended until the interval is completed. Thus, pages that are being simultaneously written by several processors do not cause thrashing. Amber [32] is a distributed shared memory system designed for a network of tightly-coupled shared memory multiprocessors. Clouds [83] provides an object-based write-invalidate distributed shared memory that is similar to Mirage in that it guarantees a writer a certain amount of time to use a page before it can be invalidated.

Spector [89, 90] implemented microcode on a Xerox Alto workstation that allowed very fast operations to be performed across an ethernet. He was able to make remote memory references in about 50 macroinstruction times. His implementation worked by using the lowest level ethernet protocols from the Alto microcode. He did not implement a transparent shared memory; special macroinstructions were needed to invoke remote references.

Munin [17, 19, 30] is a set of runtime libraries and extensions to C++ that support a distributed memory model. Munin allows the programmer to specify the expected reference pattern of an object, and then uses a coherence protocol appropriate for that pattern. In particular, only-read, initialized and then only-read, migratory, private, write-update, and synchronization objects are among those supported. Write-update is implemented by keeping an unmodified copy of the object that is marked write-update, and later comparing it against the copy that is written, sending out a list of differences to other processors having replicas of the object.

Munin uses an understanding of the locking structure of the program to implement release consistency in software. Release consistency is a weaker form of coherence (see Section 1.3.5) that operates by not requiring that remote access to a data structure get correct data until the lock protecting that data is released by its writer. This allows Munin to batch together many remote writes (including those made to a write-update object) into a single message, and thus eliminate much of the large per-message overhead that is present in networked systems.

Stumm [93] describes in detail various algorithms used to implement distributed shared memory systems.

1.3.4 Multiprocessor Tracing Techniques

The primary technique used to establish my thesis is the analysis of multiprocessor memory traces. As such, work related to collection and validation of such traces is important to understand.

There are several methods of collecting traces. One is to monitor bus transactions with auxiliary hardware, such as in [36]. Another is to single-step programs and record what happens at each step, which is the approach used in this work, and has also been used in such studies as [43] and [100]. Simulating the entire hardware architecture in software, such as in Tango [53], can be used to gather traces, although it is typically employed for developing software for a machine that is unimplemented. Another technique is to modify the microcode (presuming that one is using a writable control store machine) to record memory references, as is done in ATUM [5].

Abstract Execution (AE) [66] is a technique whereby a program is compiled into a "schema," which describes only computations necessary to determine the actual addresses referenced. This schema, together with output produced by running the program with special tracing code added by a modified version of the C compiler (which also produces the schema), is sufficient to dynamically recreate the memory references at minimal cost. The output files are typically 2-3 orders of magnitude smaller than the uncompressed trace files, and the regeneration of the traces from the output and the schema is relatively inexpensive. Unfortunately, this technique is not designed for multiprocessor usage, and would require significant adaptation for such use.

TRAPEDS [94] is a system for collecting multicomputer traces by modifying the code to emit a trace entry at the time when the reference is made; the trace analysis is carried out concurrently with the execution of the program being traced, and the actual trace entries are never saved, thus eliminating a large amount of I/O and the necessity for a large volume of storage. This scheme is useful if the amount of time spent analyzing a trace is much larger than the amount of time used to collect it, so that multiple analysis runs won't be significantly slowed by the time to regenerate the trace. Alternately, it is useful if one wants to rerun traces with different application parameters (such as number of processors, input data, or hypercube topology), and analyze each trace exactly once. TRAPEDS relies on being run on a machine that does not have remote memory references, but rather must use explicit message-passing operations to communicate; otherwise its trace interleaving would be incorrect.

MPTrace [44] is a system that uses modification of assembly code to have the executing program generate trace data as it runs, much in the fashion of Abstract Execution. Its primary concern is limiting the amount of time spent collecting the trace data, thus limiting the distortion caused by varying the relative rates of execution of instructions. MPTrace achieves a factor of only 3 slowdown, as compared to two to three orders of magnitude for single-step based methods, 20 for ATUM, 10–30 for TRAPEDS and 1.5-1.8 for (uniprocessor) AE. Koldinger *et al.* [62] examine the effects of tracing-induced slowdown on the results of a coherent caching simulation, and find that even dilation by a factor of 250 rarely produces statistically significant changes in results, thus indicating that reducing dilation is probably not critical to the validity of the simulations run on the traces.

Wisconsin's Wind Tunnel [55] is a Thinking Machines CM-5 with software that allows simulation of different memory architectures by using the error checking and correction (ECC) feature of the hardware to simulate cache misses. When a cache line would be unavailable, its data are written with their incorrect ECC bits; any reference to such a line causes a fault and trap into the operating system, which can then simulate a coherence operation. Memory references to lines that would be locally available (and of course, non-memory referencing instructions) execute at full speed.

There are also some very clever software techniques for managing and collecting traces. [87] describes Mache,[5] a scheme for compressing raw trace data by using differences in addresses and maintaining several streams. Mache is typically able to achieve a factor of 20 compression over raw, uncompressed traces.

[5]Pronounced "Mash," like "cache."

1.3.5 Weak Coherence

Sequential consistency is a basic premise of the work presented in this dissertation. That is, memory is kept up to date in such a way that all processors reading memory locations will see data that is consistent with some possible sequential ordering of updates to the shared memory. There has been a considerable amount of work done on relaxing this coherence model in order to improve system performance. The cost of this weakening is that memory semantics as presented to the runtime system (and most likely the applications programmer) are less intuitive; on the other hand the performance benefits can be considerable.

The DASH multiprocessor system [69], as well as Munin use *release consistency* [52]. In release consistency, updates are guaranteed to happen only at the time that the lock protecting a given object is released, rather than when the write is performed. Thus, any reads made to data that have been updated but whose protecting lock has not been released may or may not return the new value. In particular it is possible that a writer updates variable A and then B, and a reader reads B and then A and sees the new value for B but not for A. In a sequentially consistent machine, this would not be permitted. In DASH, there are special hardware operations called *fences* that force all writes to complete. They must be inserted by the programmer when locks are released. Since Munin is integrated with the C++ language system, it is able to tell synchronization points by examining the program, and so it doesn't need any special hardware operations to support release consistency.

2 Application Styles, Tracing and the Application Set

When comparing components of a multiprocessor memory system such as the cache line size, NUMA policy, or memory bandwidth it is helpful to vary as little else as possible. However, performance of these systems depends heavily on an external factor: the application being run on the system. Using synthetic or unrealistic applications can invalidate the results of an otherwise correct study. In practice there are very large differences in reference behavior between applications.

The application mix is fundamentally unlike factors such as the coherence policy: while the policy is a part of the system being constructed, and may be adapted with the system if it changes, the applications exist independent of the system. The system-independence of applications is mitigated somewhat by the fact that the reference behavior of an application depends on the compiler and system-supplied runtime libraries, and because programmers sometimes tune applications to machines, but by and large what the applications do and how they work is determined *a priori*. Therefore the application mix used in a trace study is of great importance, and must be selected with some degree of care.

An alternate to real applications is to use synthetically generated traces to drive the simulation. I chose not to use this method because I believe that it can very easily lead to results that are primarily dependent on the way in which the traces were generated rather than on the underlying system being studied. In particular, subtle things like false sharing can have a tremendous effect on program performance, yet are hard to describe as parameters to a trace generator (see Chapter 7 for a description of false sharing.)

This chapter contains a description of the various programming styles used to write the applications in my trace suite, the tracing method employed, and the applications themselves.

2.1 Application Programming Styles

The applications used here are written in three basic styles: C-threads, EPEX and Presto. Each style has a characteristic method of expressing parallelism and distributing data. This section describes the styles and comments on the effect they have on the applications written in them.

2.1.1 C-Threads

C-Threads [37] is the method of writing parallel C [61] programs provided with the Mach system. It is implemented as a library of C functions that build multi-threaded semantics on top of the Mach task and thread abstractions. In addition, it provides a set of primitives for synchronization of threads, some of which are implemented using the Mach interprocess communication system, and some of which use shared memory operations, primarily test-and-test-and-set based spin locks.

In a sense, C-Threads doesn't place much in the way of limitations on the style in which programs are written. More or less any type of parallelism can be expressed in C-Threads, from very fine-grained sharing with little data locality, to what amounts to a message passing system with no memory sharing at all. However, the particular C-Threads programs in the application set are in two basic categories: those written for the ACE or Butterfly, and those from the SPLASH [88] benchmark suite. The SPLASH programs are `mp3d`, `cholesky` and `water`. `Plytrace` was written specially for the ACE by Armando Garcia, `matmult` was written for the ACE by Bob Fitzgerald, and `chip` was modified to run on the ACE by Fitzgerald. The remainder of the C-Threads applications were written for the Butterfly.

The Bufferfly programs were written in the style that seemed to most naturally express the parallelism inherent in the problem [47]. Most of them implement core algorithms, rather than solving complete problems. Some of them have one large data structure to which most memory references are made, and the problem sizes are chosen so that the pieces used by any particular processor are in chunks of 4 kilobytes, the Butterfly pagesize. The natural layout for smaller data structures is such that they can generate a lot of data communications activity if they are not handled properly.

The SPLASH programs are more of a mixed bag. Two of them (`cholesky` and `mp3d`) were not written with data locality in mind, and so in general have worse performance.[1] They are not instances of programs whose parallel implementations require a lot of communication between processors, and so they could have been

[1]`Mp3d` has since been rewritten to improve locality.

written in such a way as to have better locality, and thus better performance. Water differs in that some effort was made to increase per-processor locality, and the main computation itself is more local. As a result, water performs better.

2.1.2 EPEX

EPEX [92] is a preprocessor that allows programmers to explicitly parallelize their FORTRAN programs. The main EPEX facility allows the programmer to specify that a DO loop executes in parallel, rather than serially as in normal FORTRAN. It is the programmer's responsibility to ensure that there are no cross-iteration dependencies (this stands in contrast to automatic or semi-automatic FORTRAN parallelizers such as PTOOL [6] that attempt to make this guarantee themselves).

In the event that there are known cross-iteration dependencies, the EPEX programmer is provided with synchronization calls that allow the elimination of races between iterations.

The ACE version of the EPEX preprocessor generates one thread per processor, and schedules adjacent loop iterations for the same processor, thus giving some hope of locality of reference, at least through the execution of a single parallelized loop. It does not attempt to do more complicated loop scheduling, and in particular does not attempt to do memory-conscious loop scheduling, such as that done by Markatos and LeBlanc [75].

2.1.3 Presto

Presto [21] is a system for writing parallel programs in C++ developed at the University of Washington. The Presto applications used here were written by students at Rice University in a way that expressed the application's parallelism naturally, but which paid no attention to issues of locality of reference. These applications run many more threads than there are processors on the ACE, and make no effort to partition the data in such a way that adjacent pieces are used on a single processor. This lack of locality results in a large amount of false sharing, and generally poor performance on machine models which do not have very small memory grain sizes.

The Presto programs in the application set and the Presto runtime system were ported to the ACE and traced by Alan Cox.

2.2 Applications

This section contains a brief description of each of the applications, as well as a size summary of each of the traces.

Trace	References	Private Refs	File Size	Bytes/Ref	Reads
e-fft	10.1	81.1	22	2.18	51%
e-simp	26.7	109	31	1.16	87%
e-hyd	41.3	445	67	1.62	77%
e-nasa	21.9	326	33	1.51	68%
gauss	270	0	477	1.77	83%
chip	413	0	772	1.87	68%
bsort	23.6	0	39	1.65	63%
kmerge	10.9	0	18.2	1.67	64%
plytrace	15.3	0	13	0.85	66%
sorbyc	104	0	114	1.10	79%
sorbyr	104	0	108	1.04	79%
matmult	4.64	0	9.8	2.11	97%
mp3d	19.6	0	24	1.22	57%
cholesky	38.4	0	40	1.04	77%
water	82.1	0	90.2	1.10	59%
p-gauss	21.7	4.91	30	1.38	65%
p-qsort	17.5	3.19	23	1.31	68%
p-matmult	6.81	.238	10.8	1.59	96%
p-life	55.9	8.0	91	1.63	76%

Table 2.1: Trace Sizes (In millions of data references and megabytes of filespace)

Table 2.1 enumerates the traces and specifies the size of each, as well as providing data regarding trace compression, which is discussed in section 2.3.2. Each entry gives the length of the trace in millions of references. Applications written under Presto or EPEX have regions of memory that are private to each processor. The number of memory references to these private regions is listed under the heading "Private Refs" and not included in "References." Cost computations (as described in Chapter 4) for applications that have private references in their traces are modified to count each private reference as if it were made to local memory. This is particularly important for the EPEX applications, in which a majority of references are made to private regions. The last column of the table indicates the fraction of the non-private references that were reads, as a percentage of the total number of non-private references.

2.2.1 EPEX applications

There are four applications written in EPEX FORTRAN. They are `e-fft`, `e-simp`, `e-hyd`, and `e-nasa`.

`E-fft` computes a fast Fourier transform of a a 256 by 256 array of floating point numbers. In an independent study, Baylor and Rathi analyzed reference traces from an EPEX fft program and found that about 95% of its data references were to private memory[13].

`E-simp` is a version of the simple benchmark [40], `e-hyd` is a hydrodynamics program, and `e-nasa` is a program that computes airflow over a 3 dimensional object.

The EPEX applications were collected by Bob Fitzgerald and Francesca Darema-Rogers at IBM. The authors of these programs are unknown to me. Furthermore, I don't have the sources for any but `e-nasa`, and even that is so poorly commented as to make it nearly impossible to understand. It is clearly a program that was once written on punched cards: it still has card identifiers in the right-hand columns of the source.

2.2.2 gauss

`Gauss` is a simple Gaussian elimination program. It uses a 512 by 512 array of unsigned, 4-byte integers and does not use partial pivoting. It is parallelized by dividing the array into bands of rows and assigning one band per processor. Since the bands are multiples of 4K bytes in size and are aligned, memory behavior is fairly regular.

The main loop of `gauss`, which accounts for nearly all of the references in the trace, is the following code fragment:

```
for(col = iter + 1; col < size; col++)
    a[row][col] -=a[pivot][col] * factor;
```

The loop makes 5 reads and one write. The reads are of `a`, `a[row]`, `a[row][col]`, `a[pivot]`, and `a[pivot][col]`, and the write is to `a[row][col]`. It interesting that the ACE compiler chooses to load `a` on each iteration through the loop: it is being excessively conservative in that it is unlikely that the integer write in the loop would be overwriting `a`, a pointer. The `gcc` compiler for the 68020 does not generate this load, and so has one less memory access in the main loop. Keeping `a[row]` in a register would remove another memory reference from the inner loop.

`Gauss` is by Rob Fowler.

2.2.3 chip

`Chip` is a program that attempts to minimize the amount of wire needed to connect 81 integrated circuit chips on a 9 by 9 board by moving the chips from place to place. It does this minimization by using simulated annealing.

Simulated annealing works by considering moving a chip a distance determined by a random function of the "temperature" and making that move if it results in a situation in which the state after the move is better than it was before the move. As time goes on, the system temperature is reduced, and only smaller moves are considered, which is similar to the process by which metal is annealed (hence the name). The technique is generally used as an approximation method for NP-complete optimization problems. In the run used in the trace, a total of 643552 moves were considered, and 146670 of them were accepted.

Bob Fitzgerald did much of the work to get `chip` to run on the ACE, but I do not know who is the original author.

2.2.4 bsort and kmerge

`Bsort` and `kmerge` are two parallel sorting programs. Both operate by dividing the array to be sorted into one piece per processor; each of these pieces is heapsorted by its respective processor.

After the initial sorts, the pieces are merged into a single resultant array. `Bsort` does the merge in phases: in each phase, the regions are grouped into pairs; one of the processors for the pair drops out of the merge, and the other merges the two pieces, which are then considered a single piece for the next merge phase. When all of the pieces are merged together, the array is sorted and the program exits.

`Kmerge` is similar to bsort, except that it uses a merge in which processors don't have to drop out in each phase, so as to increase parallelism in the merging process.

The number of 4 byte integers to be sorted in each program is such that they divide evenly by the number of processors (8), and furthermore when so divided, they fit evenly onto (4K) pages. There are 163840 numbers to be sorted in `bsort` and 81920 in `kmerge`. Neither of the programs uses in-place merging, so there is twice as much memory used as is needed to hold the data; in each merge phase, one copy is read and the other written. Much of the data is migratory between phases, in large chunks.

These programs are described in detail in [7]. They were coded by Rob Fowler. `kmerge` was based on an implementation by John Mellor-Crummey.

2.2.5 plytrace

Plytrace [49] is a floating-point intensive C-threads program for rendering artificial images in which surfaces are approximated by polygons. One of its phases is parallelized by using as a work pile its queue of lists of polygons to be rendered. Unfortunately, a straightforward implementation led to poor speedup, because the amount of work needed to render a given polygon varies greatly from polygon to polygon. Thus, before the parallel phase begins the list of polygons to be rendered is sorted in decreasing order of expected time-to-render, thus improving speedup.

`Plytrace` is by Armando Garcia; he wrote it to run on the ACE.

2.2.6 sorbyr and sorbyc

These programs employ successive over-relaxation [81], evaluating LaPlace's equation to compute the steady state temperature of the interior points on a piece of metal, given the temperature of the edges. The piece of metal is represented by an array of horizontal strips. Each horizontal strip is an array of doubles. This method of data allocation is used instead of a single two dimensional array so that each row could be allocated to its own page.

SOR is a red/black algorithm. Imagine the piece of metal as a checkerboard. Each point is a square, and the squares are colored red and black as on the checkerboard. Each red square's orthogonal neighbors are black, and vice versa. A particular phase of the computation updates only the red or black squares; the subsequent phase updates the other color. The phases are separated by barriers. The computation is such that the new value of a square depends only on its orthogonal neighbors. Therefore, there are no data dependencies within a red or black phase. The process of updating one color and then the other occurs a fixed number of times, and the program exits.

SOR is parallelized by splitting the piece of metal into horizontal bands (composed of several "strips" which are single array rows), with one band allocated to each processor. In the traces used here the array is 406 elements square. Since there are seven processors, each processor has a band 58 rows wide. The interesting memory behavior of the program lies in what it does near the edges of the bands. The value of the squares in an edge row depend on those just past the edge. Thus, edge pages are used by more than one processor.

The difference between `sorbyr` and `sorbyc` is the order in which they update their squares. `sorbyr` processes a row at a time, while `sorbyc` operates by columns within a band. The effect of the difference in order of computation is that the grain of sharing for the boundary rows is much more coarse in `sorbyr`, because all processors are at the top of their band (and so using the bottom of their upward neighbor's band) at the same time, while references roughly alternate in `sorbyc`.

These programs were coded by Alan Cox and Rob Fowler; I adapted them to run on the ACE.

2.2.7 matmult

`Matmult` is a simple matrix multiplication. Two source arrays are multiplied to produce a product array. The arrays are of size 200x200. As opposed to the more traditional floating point operations, `matmult` only uses integer arithmetic. This was done primarily to avoid having the execution time be dominated by the (fairly slow) floating point unit on the ACE when the program was being used for timing studies. In the trace studies the only effect of using integer rather than floating point is that integers are 32 bits, while double precision floating point numbers are 64.

`matmult` is written using the ACE parallel timing facility, which means that in the trace the multiplication is run twice on a single processor, and then once for each number of processors from seven to one. That is, there are really 9 matrix multiplications being run with differing numbers of processors (three of the runs are on just one processor).

This program was written by Bob Fitzgerald for the ACE.

2.2.8 mp3d

`mp3d` is a SPLASH program that computes rarefied fluid flow using a Monte-Carlo method. It considers the movement of an object (say, a wing) through a region of space, which is represented by a collection of cubic "cells." Movement is simulated by introducing fluid molecules at the front of the simulated region with a backward velocity. When molecules travel out the back end of the simulated space, they are recycled at the front. Molecules that occupy the same cell at the same time may collide, with results depending on the translational and rotational velocities of the molecules.

The program is parallelized by statically assigning molecules to processors. At each time step, each processor updates its molecules' location and velocity, and modifies the space array to indicate the molecules' new positions. Thus, there is some processor locality in the molecule array, but it is not allocated in such a way that each processor's piece is aligned, so there may be some false sharing around the edges of a processor's region. There is essentially no locality beyond the size of a single molecule in the space array, since any molecule (and hence the references made by its processor) may exist in any location in the simulated space.

2.2.9 cholesky

This program is from the SPLASH application set. It performs a Cholesky factorization of a sparse matrix, using the supernodal fan-out method described in [85]. The program is parallelized by use of a parallel work queue: as a supernode is processed, it may result in other supernodes becoming ready to process, and if so they are placed on the global task queue. When the queue empties, the job is complete. The only synchronization is on the task queue.

This program makes no effort to enforce data locality, so while the grain of real data communication is fairly large (it is at least the grain of a single task from the work queue) much apparent (false) sharing can occur because memory references to colocated portions of the sparse matrix are essentially random.

2.2.10 water

`Water` solves an N-body molecular dynamics problem. Specifically, it determines a variety of static and dynamic properties of liquid water. It is a C language translation of a FORTRAN program from the Perfect Club benchmark set [20]. To reduce the complexity of the N-body problem, only the forces from molecules within a certain radius of the molecule currently being updated are considered.

The main data structure is an array of per-molecule structures, containing various information describing the state of the molecule and its constituent atoms. The process is parallelized by assigning a set of molecules to a given processor; that processor computes the resultant forces on those molecules in each time step. Locality is maintained by ordering the molecules in the input stream in such a way that molecules that are spatially close to one another (and so strongly interacting) are near to one another in the input array, and so likely to be handled by the same processor.

At first blush, this program seems as if it would have locality characteristics very similar to those of `mp3d`. However, observed performance (see figures A.11 and A.12) is radically different, indicating that `water` has much greater locality of reference than does `mp3d`. Consider what happens in `mp3d`: molecules are moving rapidly through the simulated space, and there is no problem-spatial locality to each of the processors. Therefore, `mp3d`'s space array is effectively randomly accessed. `Water`, on the other hand, has molecules which move more slowly, and which are grouped problem-spatially onto processors. Furthermore, `water` does much more computation per time step, much of which is local to a particular molecule. All of these things add up to less sharing (both true and false) and better overall performance than in `mp3d`.

`Water` was modified from the SPLASH suite to run on the ACE by Cezary Dubnicki.

2.2.11 p-gauss

`P-gauss` is a Gaussian elimination program, much like **gauss**, except that its array is 100 by 100 elements, and it is parallelized by forking one thread per column instead of one thread per processor, with each thread processing a block of columns.

2.2.12 p-qsort

The quicksort algorithm [56] operates by a divide and conquer strategy: to quicksort an array segment, select a pivot element, partition the array so that all elements less than the pivot are to its right and all greater than the pivot are to its left, and then quicksort the two resulting partitions. This final recursive step provides for a natural parallelization: simply quicksort the two resultant partitions in parallel.

`P-qsort` does just this, with the exception that when the size of a partition is 10 or less, it serially bubble sorts the partition instead of recursing. In the trace used here, `p-qsort` operated on an array of size 20,000 integers, resulting in 2000 threads. These threads are essentially randomly assigned to processors, resulting in very little locality of reference, and hence bad performance for machines in which the page (or coherent cache line) size is not as small as 10 words. Performance would be greatly improved simply by placing threads on processors in such a way that adjacent sections of the array were handled by a single processor.

2.2.13 p-matmult

`P-matmult` is a parallel matrix multiply program, similar to `matmult`. The primary differences are that (1) the program is parallelized by assigning one thread for each row of the resultant array rather than one per block of resultant rows; (2) the arrays being multiplied are 100 elements square; and (3) the multiplication is run only once.

2.2.14 p-life

`P-life` is an implementation of Conway's game of life [51]. It computes 500 generations of a 42 by 42 cell world. Unlike the other Presto applications, it uses only one thread per processor, and divides the work up into bands of the array. However, since the array is so small relative to the number of processors the locality of reference is still poor.

2.3 The ACE Tracer

Trace analysis is only as valid as the traces on which it is performed. Therefore, it is important to have a good understanding of the method used to gather the traces and what effect the tracing process had on the applications being traced.

All of the traces used here were gathered on the IBM ACE Multiprocessor Workstation. The ACE is described in detail in section 3.1.1. Briefly, it is an eight processor shared bus NUMA machine in which one processor is usually dedicated to handling Unix system calls. It runs a special version of the Mach operating system; the interesting changes made to Mach for the ACE are the subject of Chapter 3.

The software used to collect the traces is known as the "tracer." It consists of two major components: a set of patches to the kernel's trap handler, and a program, `trace_saver`, that packages and saves the references obtained by the kernel from the traced application.

2.3.1 Kernel Modifications to Support Tracing

The tracer operates by single stepping all threads within the traced application, decoding each instruction executed and recording it and any memory references made by that instruction. These memory references are recorded in a buffer, and when the buffer fills `trace_saver` is awakened to empty it.

All executed instructions, regardless of whether they reference memory, are written into the kernel buffer. This prevents threads that vary in the fraction of instructions that reference memory from running at different speeds, but slows the overall execution somewhat by increasing contention for the buffer lock, and filling the trace table faster even if the recorded instruction references are later ignored by `trace_saver`.

The trace buffer consists of 2000 entries. Each entry describes a single executed instruction: the address of the instruction, the address of the memory reference made by the instruction (if any), whether that reference was a read or write, the length of the reference and whether the reference was due to DMA by the 68881 floating point processor [80]. The ACE's ROMP processor does not support any memory-to-memory operations, and restricts the number of bytes of memory affected by any one instruction to one of 1, 2, 4 or a multiple of 4 up to 64. Therefore only a single address is allocated per instruction in the buffer, and only a few bits are needed to record the length of the reference.

It is desirable to detect which references in a trace are due to spinning synchronization, because it is reasonable to believe that if the same program were run on a machine in which a remote memory access took much longer, then fewer references would occur per spinning event (though the spinning may take the same amount of time) or, more likely, synchronization would be implemented without remote spinning [77].

The ROMP processor has support for synchronization in the form of a test-and-set instruction. To detect the addresses that were targets of a test-and-set, I modified the kernel to recognize a certain otherwise invalid opcode, and modified the application program binaries to replace the normal test-and-set operation with this (otherwise) illegal opcode. When that opcode was executed, the program took a trap which was caught by the kernel. The instruction was decoded and the address referenced by the test-and-set was determined. If that address had not previously been used as the target of a test-and-set, it was printed and added to a list of already seen test and set targets. The actual test-and-set operation was executed on behalf of the user, its result stuffed into the appropriate user register and the program resumed.

It would have been easier, and preferable, to have the tracer itself print the test-and-set references when they occurred during tracing. This would have relieved me from the sometimes burdensome task of disassembling and modifying the binary of each target program. I didn't do this because most of the traces were already collected before the need for finding synchronization references became clear. Even if the tracer had produced this list, I would still have had to post-process the trace to remove references made to synchronization locations by normal instructions.

2.3.2 The trace_saver Program

The trace_saver program is the user side of the ACE tracer. Its primary function is to empty the data from the kernel's buffer, compress it, and write it into a file to be used later during trace analysis. Two types of compression are done to the trace file. First, the memory references are per-stream offset encoded. What this means is that a set of 32 addresses is kept inside **trace_saver**. These addresses are the last read, the last instruction fetch, and two different writes from each of the eight processors. Whenever a new memory reference is to be saved, if it is an instruction fetch or a read, it is saved as an offset from the appropriate stream. If it is a write, it is stored as an offset from the write stream whose address is closest to that of the write. The offsets are saved in one of four formats: 1 bit (the reference was either 2 or 4 bytes higher than the last one in the stream), 1 byte (signed), 2 bytes or a complete 4 byte address rather than an offset. This streams-based compression method was inspired by Mache [87].

To make the created trace files more manageable, they are broken into pieces automatically by **trace_saver**. These pieces are then compressed using the Unix compress facility. The final trace files for the application set vary in size from 0.8 to 2.2 bytes/reference, with the mean being about 1.4. If the instruction fetch references had been saved, the average would have been lower, because instructions tend to be very regular in their patterns, and are normally encodable in the 1-bit offset format. The compression achieved by the ACE tracer is shown in Table 2.1.

`Plytrace` has a very good compression rate. There are two reasons for this: it has very tight spatial locality in that 72% of its references were stored in the 1-bit or 1-byte format, and 55% of its references were made by one processor (many of these came during serial phases of the rendering operation). The good spatial locality results in a smaller number of bytes for compress to compress, and the large serial section improves the compression that compress achieves because the "processor number" section of the references is the same in many cases.

Conversely, `e-fft` and `matmult` used more than 2 bytes/reference. `Matmult` achieved such poor compression because its main loop performed two memory references: read a value from each of the two source arrays. These arrays were more than 32 kilobytes apart (since they are much larger than that themselves), and since the compression scheme allowed for only one read stream per processor, each of the references in the main loop generated a trace entry of the full-address format. In fact, 97% of the references in `matmult` are in the long format. `E-fft` is a different matter. While its trace is biased toward the longer offset types, it is not nearly so severe as that for matmult. However, compress did an unusually poor job on `e-fft`, resulting in the overall bad compression. Compress's poor performance is probably due to the very even distribution of reads and writes, and the even distribution of references from each processor, which resulted in a more heterogeneous input to compress, and so a worse compression ratio. However, I have not performed a more in-depth analysis of the situation to test this hypothesis.

3 Implementation of a Simple Kernel-Based NUMA System

This chapter describes an operating system kernel that implements software coherence through virtual memory mapping hardware. It was built as an initial attempt at a solution to the NUMA problem, and the lessons learned in the implementation (and, more importantly, the things that could not be determined from the implementation) served to motivate the trace studies that comprise the bulk of this dissertation.

The machine used for this work is an IBM ACE multiprocessor workstation [50]. It is an eight processor, shared bus machine with memory associated with each processor and also "global" memory that is slower to access than a processor's own memory, but faster than that of another processor. The coherence software was embedded in the machine dependent layer of the Mach [1, 84] virtual memory system, which eased the implementation greatly.

The ACE kernel has a simple mechanism to automatically assign pages of virtual memory to appropriately located physical memory. By managing locality in the operating system, it hides the details of the specific memory architecture, so that programs are more portable. It also addresses the locality needs of the entire application mix, a task that cannot be accomplished through independent modification of individual applications. Finally, it provides a migration path for application development. Correct parallel programs will run on the ACE without modification. If better performance is desired, they can then be modified to better exploit automatic page placement, by placing into separate pages data that are *private* to a process, data that are shared for reading only, and data that are *writably shared.* This segregation can be performed by the applications programmer on an *ad hoc* basis (using a system such as Munin [19]) or, potentially, by special language-processor based tools.

The strategy for page placement is simple, and was embedded in the machine-dependent portions of the Mach memory management system with only two man-months of effort and 1500 lines of code. It uses local memory as a cache for global memory, managing consistency with a directory-based ownership protocol similar to that used by Li [72] for distributed shared virtual memory. Briefly put, it replicates read-only pages on the processors that read them and moves written pages to the processors that write them, permanently placing a page in global memory if it becomes clear that it is being written routinely by more than one processor. Specifically, it assumes when a program begins executing that every page is *cacheable*, and may be placed in local memory. It declares that a page is *noncacheable* when the consistency protocol has moved it between processors (in response to writes) more than some small fixed number of times. All processors then access the page directly in global memory.

It is likely that simple techniques can yield most of the locality improvements that can be obtained by an operating system. The ACE results, though limited to a single machine, a single operating system, and a modest number of applications, support this intuition, as do the results presented in Chapter 5. The ACE kernel achieved good performance on several sample applications. For some of the others, it is unlikely that any change in the kernel policy would result in significantly better performance. Rather, the data sharing patterns of the applications need to be modified to improve performance, for example by changing the way in which data are grouped into pages in order to reduce false sharing.

The ACE NUMA management system was built in the summer of 1988, which was roughly contemporary with the the early work on PLATINUM [39] and the work that led to the DUnX system at Duke University [63]. The ACE work is distinguished by its emphasis on simplicity, and by its use of a machine that has global memory, no block transfer hardware and a relatively small penalty to access non-local memory. The impact of architectural parameters on coherence policy design is discussed in Chapter 5.

The kernel implementation is described in Section 3.1. This discussion includes a brief overview of the Mach memory management system and the ACE architecture. Performance measurements are presented in section 3.2. Section 3.3 discusses implementation experience.

3.1 Implementing Software Coherence on the ACE

The project to implement software coherent memory on the ACE was intended to be a short term endeavor. As a result, it was necessary to keep the design of the coherence system as simple as possible. At the time that the coherence work was begun, Mach version 2.0 was already running on the ACE. It dealt with the ACE's distributed memory by using the per-processor local memories only for (kernel and user) code, but no writable data. It seemed easiest to make the fewest modifications to the kernel that would still enable writable data to reside in the local memories.

There were two primary results of the limited schedule for the ACE project. First, all of the code to handle memory coherence was written as part of the machine dependent virtual memory code. This is not, in general, an inappropriate place for it, except that this design decision resulted in all coherence-generated faults going through the entire Mach machine independent fault process: checking to see if they should generate a user exception, if they necessitate a copy-on-write, and finally what the "physical" address for the faulting virtual address should be. For coherence-related faults, none of this is necessary: there should be no user exception, and no copy-on-write, and the coherence mechanism itself could much more easily keep track of the real location of pages. These things do not change the functionality of the software coherence, nor do they affect the coherence policy which is implemented. However, there is a performance penalty for doing this extra work.

The second design decision that resulted from expediency was that the local memories would be treated as a cache of the ACE's global memory. The effect of this decision is that, though a fully configured ACE contains a total of 80 Mbytes of memory (8 Mbytes at each of 8 processors and 16 Mbytes of global), to the user it appeared as if the machine had only 16 Mbytes, and if more was used it would begin paging (a truly slow operation on the ACE). The main reason for limiting the machine-independent memory to the size of the global page pool was that there was no way (at the time) to modify the Mach virtual memory system to understand a "physical" memory that changes in size over time. However, if local memories were not simply treated as caches then as the degree of replication changed, the number of machine-independent "pages" available would similarly change. Rather than modify Mach's virtual memory system, we accepted the limitation on the amount of available memory.

The rest of this section describes the ACE system architecture, the important aspects of Mach's virtual memory system, and the changes made to the Mach machine dependent virtual memory code to implement coherence.

3.1.1 The IBM ACE Multiprocessor Workstation

The ACE Multiprocessor Workstation [50] is a NUMA machine built at the IBM T. J. Watson Research Center from 1986 to 1988. Each ACE consists of a set of processor modules and global memories connected by a custom global memory bus (see Figure 3.1). Each ACE processor module has a ROMP-C processor [59], Rosetta-C memory management unit and 8Mb of local memory. Every processor can address any memory, with non-local requests sent over a 32-bit wide, 80 Mbyte/sec Inter-Processor Communication (IPC) bus designed to support 16 processors and 256 Mbytes of global memory.

Packaging restrictions prevent ACEs from supporting the full complement of

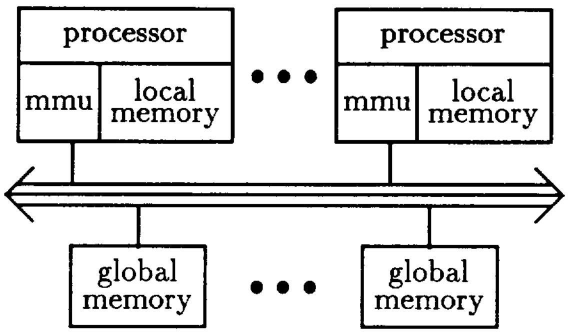

Figure 3.1: ACE memory architecture

memory and processors permitted by the IPC bus. The ACE backplane has nine slots, one of which must be used for (up to) 16 Mbytes of global memory. The other eight may contain either a processor or 16 Mbytes of global memory. Thus, configurations can range from 1 processor and 128 Mbytes to 8 processors and 16 Mbytes, with a "typical" configuration being the latter.

The measured time for a 32-bit fetch or store of local memory is 0.65μs and 0.84μs, respectively. The corresponding times for global memory are 1.5μs and 1.4μs. Thus, global memory on the ACE is 2.3 times slower than local on fetches, 1.7 times slower on stores, and about 2 times slower for reference mixes that are 45% stores. These times were measured by writing a program that did a large number of the operation in question, measuring the wallclock execution time of the program and then dividing by the number of operations. The measured times are not necessarily an even multiple of the processor cycle time because not every execution of a particular operation takes the same number cycles, because of bus and memory contention. ACE processors can also reference each other's local memories directly, but the ACE kernel does not use this facility.

The ACE does not have caches for its memories (excepting a very small on-chip instruction cache). Neither does it have special hardware to support synchronization operations: the only support provided is that the ROMP test-and-set operation operates properly on non-local memory, and 32 bit reads and writes are guaranteed to be atomic, by virtue of the bus being 32 bits wide. There is also no hardware support for a fast "block transfer" operation on the ACE bus: the fastest way to move data from one memory to another is simply to map the appropriate memories and use the C library `bcopy(3)` routine.

Since the bus has such high bandwidth and low latency relative to the processors,[1] there is little bus contention on the ACE. Furthermore, there is little time spent contending for the global memory. This is borne out by changing from a single-bank global memory to a two bank memory, and observing that the execution time

[1]ACE processors benchmark at 4320 dhrystones/second using dhrystone 1.1.

of programs on the ACE was essentially unchanged. If there had been much contention for the memory, then adding a second memory bank should have roughly halved the time spent waiting due to contention. Since this did not happen, it appears that contention is not a major contributor to execution time on the ACE.

3.1.2 The Mach Virtual Memory System

Perhaps the most important novel idea in Mach is that of machine-independent virtual memory [84]. The bulk of the Mach VM code is machine-independent and is supported by a small machine-dependent component, called the *pmap layer*, which is intended to manage address translation hardware including such things as translation lookaside buffers (TLBs), TLB reload mechanisms, and page tables. The *pmap interface* separates the machine-dependent and machine-independent parts of the VM system.

A Mach *pmap* (physical map) is an abstract object that holds virtual to physical address translations, called *mappings*, for the resident pages of a single virtual address space, which Mach calls a *task*. The pmap interface consists of such pmap operations as *pmap_enter*, which takes a pmap, virtual address, physical address and protection, and maps the virtual address to the physical address in the given pmap with the given protection; *pmap_protect*, which sets the protection on all resident pages in a given virtual address range within a pmap; *pmap_remove*, which removes all mappings in a virtual address range in a pmap; and *pmap_remove_all*, which removes a single physical page from all pmaps in which it is resident. Other operations create and destroy pmaps, fill pages with zeros, copy pages, etc. The protection provided to the pmap_enter operation is not necessarily the same as that seen by the user; Mach may reduce privileges to implement copy-on-write or the external paging system [103].

A pmap is a cache of the mappings for an address space. The pmap manager may drop a mapping or reduce its permissions, e.g. from writable to read-only, at almost any time. This protection reduction may subsequently cause a page fault, which will be resolved by the machine-independent VM code resulting in another *pmap_enter* of the mapping. This feature had already been used on the IBM RT/PC, whose memory management hardware only allows a single virtual address for a physical page, and must drop the old mapping for a physical page when it is mapped into a second address space. On the ACE, reduction of permissions and dropping of mappings is used to drive the coherence protocol. While it would be possible to determine if a given fault was due to a coherence protection reduction before handing it off to Mach's fault handler and so avoid calling Mach at all, this would have added unnecessary complexity to the (prototype) system. In a production quality system, we would have done this short-circuiting to save time. Nevertheless, Mach's ability to handle such "unexpected" faults without complaint considerably eased the coding effort.

Mappings can be dropped, or permissions reduced, subject to two constraints. First, to ensure forward progress, a mapping and its permissions must persist long enough for the instruction that faulted to complete. Second, to ensure that the kernel works correctly, some mappings must be permanent. For example, the kernel must never suffer a page fault on the code that handles page faults. Because of this second constraint, the ACE coherence system does not handle kernel memory: all kernel data pages are wired in global memory. Again, in a production quality system this decision would have to be revisited.

Mach views physical memory as a fixed-size pool of pages. It treats this pool as if it were real memory with uniform memory access times. It is understood that in more sophisticated systems these "machine independent physical pages" may represent more complex structures, such as pages in a NUMA memory or pages that have been replicated. Unfortunately, at the time that the ACE kernel was implemented there was no provision for changing the size of the page pool dynamically, so the maximum amount of memory that can be used for page replication must be fixed at boot time.

3.1.3 The ACE pmap layer

The pmap layer for the ACE is composed of 4 modules (see Figure 3.2): a pmap manager, an MMU interface, a NUMA manager and a NUMA policy. Code for the first two modules was obtained by dividing the pmap module for the IBM RT/PC into two modules, one of which was extended slightly to form the pmap manager, and the other of which was used verbatim as the MMU interface. The pmap manager exports the pmap interface to the machine-independent components of the Mach VM system, translating pmap operations into MMU operations and coordinating operation of the other modules. The MMU interface module controls the Rosetta hardware. The NUMA manager maintains consistency of pages cached in local memories, while the NUMA policy decides whether a page should be placed in local or global memory. There is only one policy, which seems to work well for many applications, but should the need arise it would be easy to substitute another policy without modifying the NUMA manager.

The NUMA Manager

ACE local memories are managed as a cache of global memory. The Mach machine-independent page pool, which is called *logical* memory, is the same size as the ACE global memory. Each page of logical memory corresponds to exactly one page of global memory, and may also be cached in the local memory of one or more processors. A logical page is in one of three states:

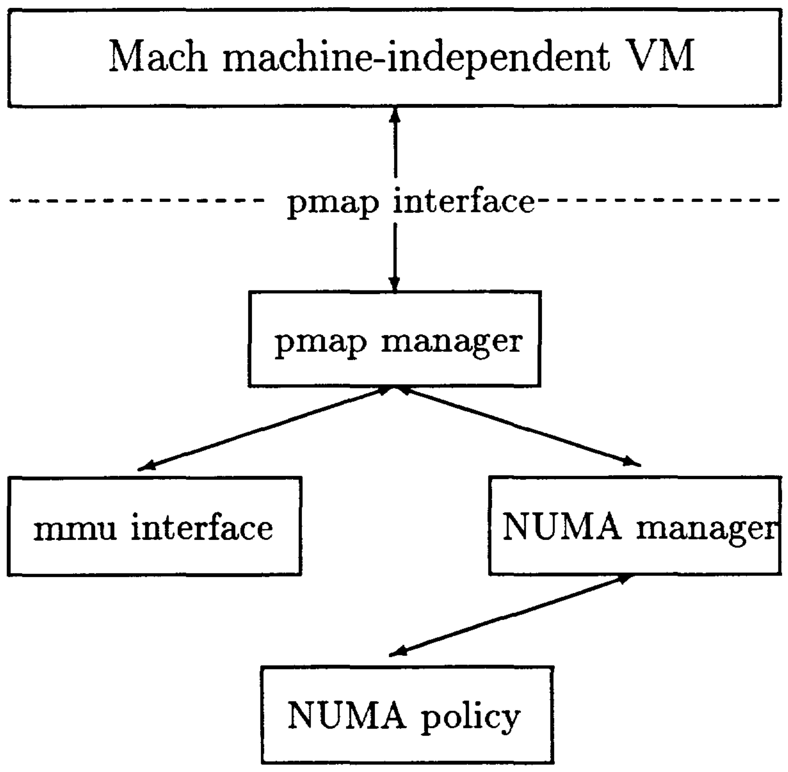

Figure 3.2: ACE pmap layer

- *read-only* — may be replicated in zero or more local memories, must be protected read-only. Global copy is up to date.
- *local-writable* — in exactly 1 local memory; may be writable. Global copy is stale.
- *global-writable* — only in global memory; may be writable by zero or more processors.

It is important to keep in mind the different levels in which protection operates in this system, and the interplay between them. A page is *user read-only* if the user has specified to the kernel that any write operations to that page should cause an exception, which will normally result in the program failing. Pages containing code (instructions) are the most common member of this class. A page is *VM read-only* if the Mach machine independent virtual memory system has decided that the page should be read-only, or if the page is user read-only. This occurs either when the page is user read-only or when Mach has the page copy-on-write. A page is *coherence read-only* when the ACE coherence system determines that the page should not be writable in order to maintain memory coherence, or the page is VM read-only. These three levels of protection form a hierarchy: all user read-only pages are VM read-only, and all VM read-only pages are coherence read-only.

A page that is VM writable (ie., not VM read-only) may go through various logical page states. It may be read by several processors for a time, and be coherence read-only. Then, it may be written by a sequence of processors and be local-writable in the memory of each writer in turn. Finally, it may become global-writable and be shared read/write by any processors that use it. These transitions are driven by the behavior of the processors using the page and the choices of the coherence policy.

The policy is implemented by the NUMA policy module through its interface to the NUMA manager. This interface consists of a single function, *cache_policy*, that takes a logical page and protection and returns a location: LOCAL or GLOBAL. Given this location, the current known state of the page, and whether the faulting reference was a read or a write, the NUMA manager then takes the action indicated in figure 3.3. The three states in the finite state machine correspond to the the possible logical page states. The transitions are labeled with 1) whether the transition should be followed on a read, write or both, and 2) the policy choice needed to invoke the transition. In the case of transitions out of the Local Writable state, the arcs are additionally labeled with whether the reference was local or remote.

These NUMA manager actions are driven by requests from the pmap manager, which are in turn driven by page faults handled by the machine-independent VM system. These faults occur on the first reference to a page, when access is needed to a page unmapped (or marked read-only) by the NUMA manager, or when a mapping is removed due to the memory management chip's restriction of only one virtual address per physical page per processor.

When uninitialized pages are first touched, Mach fills them with zeros in the course of handling the initial zero-fill page fault. It does this by calling the *pmap_zero_page* operation of the pmap module. The page is then mapped using pmap_enter and processing continues. Since the processor using the page is not known until pmap_enter time, the ACE kernel lazy-evaluates the zero-filling of the page to avoid writing zeros into global memory and immediately copying them.

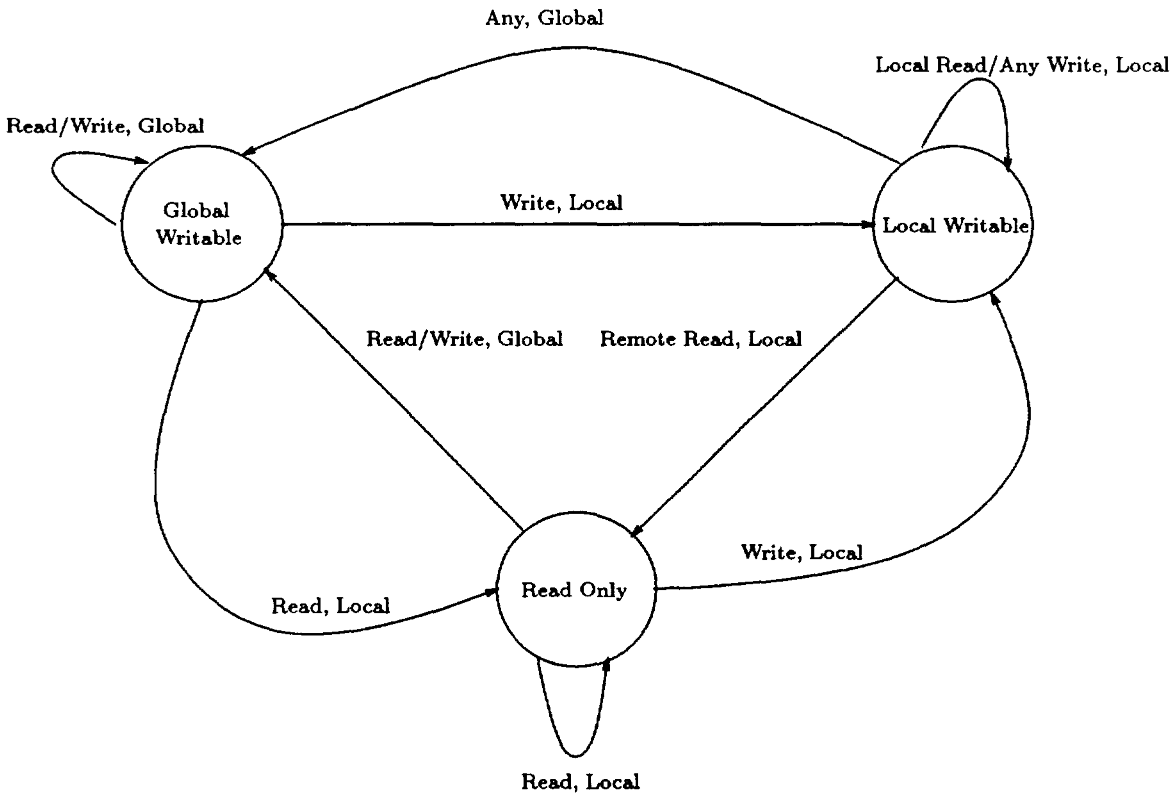

Figure 3.3: NUMA manager actions to maintain coherence

A Simple NUMA Policy That Limits Page Movement

In the ACE pmap layer, the NUMA policy module decides whether a page should be placed in local or global memory. There is only one implemented policy (other than those used to collect baseline timings; see section 3.2). It operates by limiting the number of times a page may be invalidated. When a page is referenced, it is replicated or migrated to the processor making the reference, until it has been invalidated a certain number of times, at which point it is frozen in global memory. Read-only pages are thus replicated and private writable pages are moved to the processor that writes them. Pages that are shared and written are eventually detected as such and frozen in global memory. The default number of invalidations allowed before freezing a page is four.

In terms of Figure 3.3, the policy answers LOCAL for any page that has not used up its threshold number of page moves and GLOBAL for any page that has. Once the policy decides that a page should remain in global memory, the page is said to be *pinned*.

Changes to the Mach pmap Interface to Support NUMA

Experience with Mach on the ACE confirms the robust design of the Mach pmap interface. Although the machine-independent paging interface was not designed to support NUMA architectures, the automatic NUMA page placement policy was implemented within the pmap layer with only three small extensions to support pmap-level page caching:

- pmap_free_page operations,
- min-max protection arguments to pmap_enter, and
- a target processor argument to pmap_enter.

The original machine-independent component of the Mach VM system did not inform the pmap layer when physical page frames were freed and reallocated. This notification is necessary so that cache resources can be released and cache state reset before the page frames are reallocated. The notification is split into two parts to allow lazy evaluation. When a page is freed, pmap_free_page starts lazy cleanup of a physical page and returns a tag. When a page is reallocated, pmap_free_page_sync takes a tag and waits for cleanup of the page to complete. Since Mach usually keeps a reasonable sized pool of free pages, and manages this pool as a FIFO queue, the pmap_free_page_sync operation normally does not need to block, because ample time has elapsed since the pmap_free_page was issued.

The second change to the pmap interface was to add a second protection parameter to pmap_enter. Pmap_enter creates a mapping from a virtual to a physical address. As originally specified, it took a single protection parameter indicating whether the mapping should be writable or read-only. Since machine-independent code uses this parameter to indicate what the user is legally permitted to do to the page, it can be interpreted as the maximum (loosest) permissions that the pmap layer is allowed to establish. The new parameter indicates the minimum (strictest) permissions required to resolve the fault. The ACE pmap module is therefore able to map pages with the strictest possible permissions—replicating otherwise writable pages that are not being written by making them read-only for the purpose of maintaining coherence. Subsequent write faults will make such pages writable after first eliminating replicated copies. If the change to pmap_enter were added to the Mach specification, the pmap layers of non-NUMA systems could avoid these subsequent faults by initially mapping with maximum permissions.

The third change to the pmap interface reduced the scope of the pmap_enter operation, which originally added a mapping that was available to all processors. Mach depended on this in that it did not always do the pmap_enter on the processor where the fault occurred and might resume the faulting process on yet another processor. Since NUMA management relies on understanding which processors are accessing which pages, it was necessary to eliminate the creation of mappings on processors that did not need them. Therefore, a parameter was added to pmap_enter that specified which processor needed the mapping.

3.2 Performance Results

Given the implementation of the ACE NUMA kernel as previously described, it is desirable to be able to measure how well it is doing. Several sorts of questions are apparent: is the NUMA policy well chosen? Is it worth switching policies for different applications? Was it wise to avoid the use of the ACE's remote reference capability?

3.2.1 Evaluating Page Placement

The primary goal here is to determine the effectiveness of the NUMA policy at placing pages in the more appropriate of local and global memory. Since most reasonable NUMA systems will replicate read-only data and code, only writable data is considered. Consider the following definitions of execution time:

- T_{numa} is total user time (process virtual time as measured by the Unix time(1) facility) across all processors when running the ACE NUMA strategy.
- $T_{optimal}$ is total user time when running under a page placement strategy that minimizes the sum of user and NUMA-related system time using future knowledge.

- T_{local} is the total user time of the application, were it possible to place all data in local memory.
- T_{global} is total user time when running with all writable data located in global memory.

T_{numa} is easily measured by running the application under the normal NUMA policy. T_{global} is measured by using a specially modified NUMA policy that places all data pages in global memory.

It would have been best to compare T_{numa} to $T_{optimal}$, but it was not possible to measure the latter in a real implementation (though the trace simulations in the latter chapters attempt to do just this), so it was compared to T_{local} instead. T_{local} is less than $T_{optimal}$ because references to shared data in global memory cannot be made at local memory speeds. T_{local} thus cannot actually be achieved on more than one processor in most cases. T_{local} was measured by running the parallel applications with a single thread on a single processor system, causing all data to be placed in local memory. Single threaded cases were run because many of applications synchronize their threads using non-blocking spin locks. With multiple threads time-sliced on a single processor, the amount of time spent in a lock would be determined by this time-slicing, and so would invalidate the user time measurements. The applications that spend much time contending for locks (such as `sorbyr` and `sorbyc`) are not included in this analysis.

Several members of the application set are not amenable to this type of analysis, and so are not included here. For example, `bsort` operates in two phases: a sort phase and a merge phase. The amount of work done in the merge phase depends on the number of processors doing the sorting. Therefore, the T_{local} run would do a different amount of work from the T_{global} and T_{numa} runs, and so be incomparable to them. `Chip` is a simulated annealing program, and so is non-deterministic; it would also produce incomparable single- and multi-processor runs. Applications with these sorts of problems are not included in Table 3.1.

A simple measure of page placement effectiveness, called the "user-time expansion factor," γ,

$$T_{numa} = \gamma T_{local}, \tag{3.1}$$

can be misleading. A small value of γ may mean that the page placement did well, that the application spends little of its time referencing memory, or that local memory is not much faster than global memory. To better isolate these effects, consider the following model of program execution time:

$$T_{numa} = T_{local}\left\{(1-\beta)+\beta\left[\alpha+(1-\alpha)\frac{G}{L}\right]\right\} \tag{3.2}$$

L and G are the times for a 32-bit memory reference to local and global memory, respectively. As noted in section 3.1.1, G/L is about 2 on the ACE (2.3 if all references are fetches, 1.7 if they are stores).

This model incorporates two sensitivity factors:

- α is the fraction of references to writable data that were actually made to local pages while running under the ACE NUMA page placement strategy.
- β is the fraction of total user run time that would be devoted to referencing writable data if all of the memory were local.

α resembles a cache hit ratio: "hits" are references to local memory and "misses" are references to global memory. α measures both the use of private (non-shared) memory by the application and the success of the NUMA strategy in making such memory local. A "good" value for α is close to 1.0, indicating that most references are to local memory. A smaller value indicates a problem with the placement strategy or, as typically found in practice, heavy use of shared memory in the application. If it were possible to use $T_{optimal}$ instead of T_{local} in Equation 3.2, then an α of less than 1.0 would only measure errors in the placement rather than the effects of sharing by the application.

β measures how much an application uses (non-instruction) memory, as opposed to operating in registers, branching and so forth. It depends both on the fraction of instructions that reference writable memory and on the speed of the instructions that do not (which itself is a function of local memory performance and of processor speed). Small values of β indicate that the program spends little of its time referencing writable memory, so the overall run time is relatively unaffected by α.

In the T_{global} runs, all writable data memory references were to global memory and none to local, thus α for these runs was 0. Substituting T_{global} for T_{numa} and 0 for α in equation 3.2 yields a model of the all-global runs:

$$T_{global} = T_{local}\left\{(1-\beta) + \beta\frac{G}{L}\right\} \tag{3.3}$$

Solving equation 3.3 simultaneously with equation 3.2 for α and β yields:

$$\alpha = \frac{T_{global} - T_{numa}}{T_{global} - T_{local}} \tag{3.4}$$

$$\beta = \left(\frac{T_{global} - T_{local}}{T_{local}}\right)\left(\frac{L}{G-L}\right) \tag{3.5}$$

The use of total user time eliminates the concurrency and serialization artifacts that show up in elapsed (wall clock) times and speedup curves. The goal of this analysis is to evaluate the page placement strategy; that strategy is not responsible for speedup effects, which by and large are determined by how the programmer codes the program. The greatest weakness of the model is that, because it uses

T_{local} rather than $T_{optimal}$, it fails to distinguish between global references due to placement "errors", those due to legitimate use of shared memory, and those due to false sharing. The trace studies described in Chapter 5 distinguish placement errors from (true and false) sharing, and evaluate the success of the ACE policy on the entire application set.

On the whole, the evaluation method was both simple and informative. It was within the capacity of the ACE timing facilities[2]. It was not unduly sensitive to loss of precision despite the arithmetic in Equations 3.4 and 3.5. It did require that measurements be repeatable, so applications such as the simulated annealing in `chip` were beyond it. It also required that measurements not vary too much with the number of processors. Thus applications had to do about the same amount of work, independent of the number of processors, and had to be relatively free of lock, bus or memory contention.

3.2.2 The Application Programs

This section presents the measurements and computed values for the applications that were amenable to alpha/beta analysis. In addition to applications included in the trace suite of Chapter 2, several others are used in the analysis of the ACE. Those applications are described here; the rest are explained in section 2.2. The extra applications are three prime finders (Primes1–3), a program designed to spend all of its time referencing global memory (Gfetch) and one designed not to reference memory at all (ParMult). They are not used in the rest of this dissertation because Gfetch and ParMult are just intended as calibration tools, while Primes1–3 were insufficiently interesting to warrant tracing and further analysis.

ParMult and Gfetch are at the extremes of the spectrum of memory reference behavior. ParMult does nothing but integer multiplication. Its only data references are for workload allocation and are too infrequent to be visible through measurement error. Its β is thus 0 and its α irrelevant. The Gfetch program does nothing but fetch from shared virtual memory. Loop control and workload allocation costs are too small to be seen. Its β is thus 1 and its α is 0. These two programs are included primarily as a sanity check on the program performance model, Equation 3.2.

The three primes programs use different parallel approaches to finding the prime numbers between 1 and 10,000,000. Primes1 [16] determines if an odd number is prime by dividing it by all odd numbers less than its square root and checking for remainders. It computes heavily (division is expensive on the ACE) and most of its memory references are to the stack during subroutine linkage. Primes2 [29] divides each prime candidate by all previously found primes less than its square root. Each thread keeps a private list of primes to be used as

[2]The only clock on the ACE is a 50Hz timer interrupt.

Application	T_{global}	T_{numa}	T_{local}	α	β	γ
ParMult	67.4	67.4	67.3	na	.00	1.00
Gfetch[3]	60.2	60.2	26.5	0	1.0	2.27
matmult[3]	82.1	69.0	68.2	.94	.26	1.01
Primes1	18502.2	17413.9	17413.3	1.0	.06	1.00
Primes2	5754.3	4972.9	4968.9	.99	.16	1.00
Primes3	39.1	37.4	28.8	.17	.36	1.30
e-fft	687.4	449.0	438.4	.96	.56	1.02
gauss	636.9	415.6	411.6	.98	.55	1.01
e-nasa	1388	925	870	.89	.60	1.06
e-hyd	4582.9	3333.3	2945.3	.76	.56	1.13
plytrace	56.9	38.8	38.0	.96	.50	1.02

Table 3.1: Measured user times in seconds and computed model parameters.

divisors, so virtually all data references are local. It also computes heavily, but makes local data references to fetch potential divisors.

The primes3 algorithm is a variant of the Sieve of Eratosthenes, with the sieve represented as a bit vector of odd numbers in shared memory. It produces an integer vector of results by masking off composites in the bit vector and scanning for the remaining primes. It references the shared bit vector heavily, fetching and storing as it masks off bits representing composite numbers. It also computes heavily while scanning the bit vector for primes.

Baylor and Rathi's study of an EPEX fft program showed that about 95% of its data references were to private memory [13]. Although there are differences in compilers and runtime support, I think that this supports the contention that the ACE NUMA strategy placed pages effectively and that the remaining global memory references are due to the algorithm in the application program.

3.2.3 Results

The measured times and computed parameters are presented in Table 3.1. Overall, the α and γ values are remarkably good, the exceptions being the Gfetch program, which was designed to be terrible, and the Primes3 program, which makes heavy legitimate use of writably shared memory. There is little hope that programs with such heavy sharing can be made to perform better on NUMA machines without restructuring to reduce their use of writably shared data.

[3]Since Gfetch and **matmult** do almost all fetches and no stores, their computations were done using 2.3 for G/L. The other applications used G/L as 2 to reflect a reasonable balance of loads and stores.

Application	S_{numa}	S_{global}	ΔS	T_{numa}	$\Delta S / T_{numa}$
IMatMult	4.5	1.2	3.3	82.1	4.0%
Primes1	1.4	2.3	na	17413.9	0%
Primes2	29.9	8.5	21.4	4972.9	0.4%
Primes3	11.2	1.9	9.3	37.4	24.9%
FFT	21.1	10.0	11.1	449.0	2.5%
gauss	31.5	16.6	14.9	415.6	3.6%

Table 3.2: Total system time for runs on 7 processors.

Of the "real" applications here (`e-nasa`, `e-hyd` and `plytrace`), only `e-hyd` made more than 10% of its data references to global memory, and it made only 24% of references non-locally. The trace based analysis presented in Chapter 5 and displayed in Figure 5.1 confirm that the slowdown seen here was due to application behavior rather than to errors made by the ACE NUMA policy.

Of course, these timings do not indicate that the ACE policy is useful on machines unlike the ACE, and in Chapter 5 I conclude that on other machines different policies perform better. Nevertheless, the results here indicate that the ACE policy did a very good job of choosing placements.

3.2.4 Page Placement Overhead

The results in Section 3.2.3 reflect only the time spent by the applications in user state, and not the time the NUMA manager uses for page movement and bookkeeping overhead. Table 3.2 shows the difference in system time between the all-global and NUMA managed cases. This difference is interesting because system time includes not only page movement, but also system call time and other unrelated overhead; since the all global case moves no pages, essentially no time is spent on NUMA management, while the system call and other overheads stay the same. Comparing this difference with the NUMA managed user times in Table 3.1 shows that for all but Primes3 the overhead was small. Primes3 suffers from having a large amount of memory almost all of which winds up being placed in global memory, but which is first copied from local memory to local memory several times. Since the sieve runs quickly, this memory is allocated rapidly relative to the amount of user time, resulting in a high system/user time ratio. Primes3 would have profited from a reduction in the number of invalidations allowed before freezing.

For some reason, `e-nasa` and Primes1 used less system time in the NUMA-managed case than in the all-global case. This may have been because user memory that the kernel needed to access was in local memory in the NUMA case, and so these accesses were faster, thus using less system time.

The ACE NUMA management code was not written with an eye towards efficiency.

3.3 Discussion

3.3.1 The Two-Level NUMA Model

Supporting only a single class of shared physical memory (global memory) was the most important simplification of the NUMA management problem. The choice to use a two-level NUMA memory model (with only local and global memory, forgoing remote reference) was motivated because there was an obvious correspondence between it and a simple application memory model that had only private and shared virtual memory. Essentially, it was possible to determine that memory was either only-read, or was private to one processor and place it appropriately. While permanent pinning of all other memory in not always the best thing to do, it is likely to be correct often, and not terribly bad in other cases, because the ACE has such a low local/global access time ratio.

The experience with Mach on the ACE has been heartening with respect to this decision. Most of the energy spent writing applications was directed toward debugging application errors, rather than worrying about memory system quirks. Automatic page placement worked well enough and predictably enough that often it could be ignored, and when it mattered it could be controlled by careful coding.

3.3.2 Making Placement Decisions

There is a fundamental problem with locality decisions based on reference behavior: it is hard to make a good decision quickly. A placement strategy should avoid pinning a page in global memory on the basis of transient behavior. On the other hand, it should avoid moving a page repeatedly from one local memory to another before realizing that it should be pinned, lest the time spent moving pages become significant. In a system such as the ACE, with its very low ratio of global reference cost to page move cost, reconsidering a pinning decision by freeing a page and letting it move again is unlikely to be beneficial. On the other hand, on machines with lower block move/remote reference time ratios, making extra moves in order to reduce remote references can be necessary; see Chapter 5 for a detailed discussion and illustration of these issues.

Any locality management system implemented solely in the operating system must suffer some thrashing of writably shared pages between local memories. This cost is probably acceptable for a limited number of small data objects, but may be objectionable when a significant number of pages is involved. For data that are known to be writably shared (or that used to be writably shared but can now be replicated), thrashing overhead may be reduced by providing placement pragmas to application programs. Such pragmas would cause a region of virtual address space to marked cacheable and placed in local memory, or marked non-cacheable and placed in global memory. These pragmas have not been implemented in the ACE kernel, but easily could be.

3.3.3 Mach as a Base for NUMA Systems

Aside from the problem of the fixed-sized logical page pool, which forces a fixed degree of replication, Mach supported the ACE effort very well. It allowed the construction of a working operating system for a novel new machine in a very short period of time. The number of changes necessary in the system was surprisingly small, considering that the pmap interface was not designed with the NUMA problem in mind. Of particular use was the ability to drop mappings or tighten protections essentially at whim; this made it unnecessary to determine which faults were caused by the NUMA-related tightening of protections and which were caused by other things.

One problem with version 2.0 Mach is that most of the Unix compatibility code is still in the kernel. The Mach project at CMU has since removed this code from the kernel, and implemented it as a set of parallel user tasks. When the ACE port was done, however, Mach implemented the portions of Unix that remained in the kernel by forcing them to run on a single processor, called the "Unix Master." This caused two problems, one for all multiprocessors and one only for NUMA systems. The first is that execution of system calls produced a bottleneck on the master processor. The second is that some of these system calls reference user memory while running on the master processor. It would be difficult and probably unwise to treat these references differently from all the others. Thus pages that are used only by one thread (stacks for example), but that are referenced by Unix system calls, can be shared writably with the master processor and can end up pinned in global memory. Several of the system calls that most caused this problem (`sigvec`, `fstat` and `ioctl`) were modified in an *ad hoc* manner to cause their user memory references to happen on the calling processor rather than the master.

3.4 Whence from Here?

The major findings of the ACE project were that:

- the simple page placement strategy worked about as well as any operating system level strategy could have on the ACE,
- this strategy could be implemented easily within the Mach machine-dependent pmap layer, and
- the dominant remaining source of avoidable performance degradation was *false sharing*, which could be reduced by improving language processors or by tuning applications.

The ACE automatic page placement was an adequate tool for coping with a NUMA memory system. It presented applications with a simple view of virtual memory that was not much harder to program than the flat shared memory of a traditional UMA multiprocessor. Correct application programs ran correctly and could then be tuned to improve performance. The placement strategy was easy to predict, it put pages in appropriate places, and it ran at acceptable cost.

False sharing is an accident of colocating data objects with different reference characteristics in the same virtual page, and is thus beyond the scope of operating system techniques based on placement of virtual pages. Because shared pages are placed in global memory, false sharing causes memory references to be made to slower global memory instead of faster local memory. Experience shows false sharing could be reduced, often dramatically, by tuning application code. Additional work is needed at the language processor level to make it easier to reduce this source of performance degradation. It would also be beneficial to be able to quantify this effect without resorting to *ad hoc* examination of the application.

This chapter mentioned several areas that need further attention. Chief among these is false sharing and what language processors can do to automate its reduction. Trace-driven analyses provide much more detailed understanding than what has been garnered through the processor-time based approach described in Section 3.2. Processor scheduling on NUMA machines is beyond the scope of this work, but is clearly important. It has been explored by Markatos [74, 75]. Similarly, having applications provide pragmas to improve the quality of the placement and to allow the use of other features such as remote reference and software implemented write-update, as well as lowering the overhead of automatic placements can be profitable, as is demonstrated by the Munin distributed shared memory system [18, 19, 30].

The comparison of alternative policies for NUMA page placement is a topic of current research [39, 58, 63, 64]. It is tempting to consider ever more complex policies, but the ACE work and that of others [65] suggests otherwise. It is unlikely that substantial additional gains can be obtained in the operating system. In some applications, or on other architectures, it is worthwhile periodically to reconsider the decision to pin a page. It may also be worth designing a virtual memory system that integrates page placement more closely with pagein and pageout, or that attempts to achieve the simplicity of the ACE cache approach without requiring that all pages be backed by global memory, or that local memory be used only as a cache [67].

The operating system itself is a parallel application worthy of investigation. The ACE project did not attempt to place any kernel data structures in local memory, other than those required to be there by the hardware. Increasing the autonomy of the kernel from processor to processor and making private any kernel data that need not be shared should both improve performance on small multiprocessors and improve scalability to large ones. Chaves *et al* explored this issue in [33].

A method of achieving answers to many of these questions is to move from real implementations to a trace-based methodology. While it is more cumbersome and less immediately rewarding than systems building, it has two major advantages. First, it allows simulation of architectures which are not easily obtained, or which do not exist at all, without major capital expenditure. Second, they can provide a deeper insight into what happens during the execution of a program than is possible in implementations. That is, it is much easier to instrument a trace simulation than a piece of hardware.

The following chapters describe a trace-based model of program execution in a NUMA system, and use that model to explore the questions of policy design, false sharing and future architectural directions in shared memory multiprocessors.

4 A Model of Program Execution in Shared Memory Systems

While the study in Chapter 3 is satisfying in as much as it describes an implementation of a real NUMA system, and offers observations about how to go about such an implementation, it was unable to answer many important questions about the ACE kernel. Discovering the fraction of references made to local memory by using Equation 3.4 is helpful only if most of the references were local; if they were not it may have been because of errors by the policy or sharing by the application. Was the 76% local hit rate good for `e-hyd`, or was it due to placement errors by the ACE policy? Could `plytrace` have done even better than its 96% hit rate? How would these applications run on architectures other than the ACE? If they were run on other architectures, would it be better to have a different NUMA policy?

It is impractical to answer these sorts of questions using implementations. Implementing kernels takes a large amount of time. Machines on which to do these implementations may not exist, or may be unavailable. Even if the machines and time for kernel implementations were available, it still would be impossible to give a good answer to questions of the type "How much better could this be?" Furthermore, as was seen in Chapter 3, many applications have behavior that prevents easily measuring their hit ratio by analyzing timings.

This leaves detailed simulation, trace-driven simulation, and analysis as the primary methods of answering the questions. A detailed simulation at the level of instructions can be difficult to construct, though results from it can be of comparable validity to those from an implementation while being able to measure things like local hit ratio directly for all applications. Detailed simulations also suffer some of the problems of implementations: they don't help with the "How good could it be?" problem, and are even slower than trace-driven simulations. On the other hand, analysis either requires a very good understanding and description of application behavior, or could reach conclusions that are not true of typical applications.

For these reasons the rest of the work in this dissertation is based on trace-driven models. Since traces can be collected from any available application, there is no need to worry that the results of the simulations are biased by a bad choice for "typical" behavior (though there could be questions about the choice of applications). It is easy to extract detailed information from trace-driven simulations, similar to what may be obtained from more detailed instruction-level simulation.

In addition, the technique of *offline optimal analysis* allows the trace-driven model to answer "How good could it be?" questions. The idea of offline optimal analysis is to determine what the decisions and performance of the best conceivable policy would be on a particular architecture and application trace. That is, it minimizes the overall placement cost for a particular architecture and application trace. To do so, it requires use of future knowledge, and is therefore not a way to build systems. However, it has great value as a technique for comparing performance. Specifically:

1. Off-line optimal analysis allows the evaluation of hardware design decisions without biasing the results based on the choice of policy (none of which can be optimal over the entire design space).

2. Performance results obtained with off-line analysis provide a tight lower bound, for a given hardware base, on the cost savings that can be achieved by any "real-life" policy. Rather than saying how much time is being used by a given policy, they say how much time must be used by *any* policy. The difference between optimal performance and actual achieved performance is the maximum time that could possibly be saved through policy improvements.

It is generally accepted that memory reference traces need to run into tens or even hundreds of millions of references to capture meaningful behavior. Any algorithm to compute an optimal set of memory management decisions must therefore make a very small number of passes over the trace—preferably only one. The optimal policy described here runs in time proportional to the number reads in the trace, and the number of writes times the number of processors.[1] Later chapters use optimal analysis extensively to evaluate policies for particular architectures and to compare different architectures without introducing policy bias.

This chapter presents the model of machines, traces, policies and cost of execution. It then describes the optimal policy, and provides pseudocode for it. It also describes a study designed to establish confidence that the trace collection process did not significantly distort application behavior.

Much of the work presented in this chapter appeared in [26].

[1]Paul Dietz has found an algorithm that is linear in the length of the trace, without regard to the number of processors.

4.1 A Model Of Memory System Behavior

This section describes a model designed to capture the latency-induced cost of memory access and data placement in a multiprocessor memory system that embodies a tradeoff between replication, migration and (possibly) single-word reference. This model describes most NUMA machines and many coherently cached machines as well. It does not attempt to capture any notion of elapsed wall-clock time, nor does it consider contention, either in the memory or in the interprocessor interconnection. Think of it as total work to reference memory. Memories and caches are assumed to be as large as needed. Instructions are assumed to be local at all times; instruction fetches are ignored.

It is important to keep in mind the intent of this idea of cost. Roughly, cost is analogous to what Mach reports as "user time" when a parallel program is run (the sum of time spent in user state for all processors), except that it only considers the portion of the time that is due to memory references, and not that due to register to register operations, branches, I/O, and so forth. The intent of the cost model is to give some insight into how much time was spent referencing memory by a program on a particular machine running a particular policy. As such, it is essentially useless for comparisons of different programs, or of versions of the same program using a different algorithm. Furthermore, cost is unrelated to parallel speedup, processor utilization or similar ideas; it is entirely possible that a program with a higher "cost" will run faster by having a higher degree of parallelism.

The caveats about what cost does not mean are even more important when cost is divided by the number of references to yield Mean Cost Per Reference (MCPR), as I have done throughout much of this dissertation. While any change in NUMA policy that reduces the MCPR of a given program running on a given machine represents a true improvement, if the program itself were simply modified to make a large number of unneeded references to private memory, its MCPR would decrease, but clearly the program would not have been improved. Conversely, if a program were changed so that some unneeded private references were removed, its MCPR would increase, but the program might nevertheless run better.

Those provisos in hand, consider what cost is able to do. When the application is unchanged, cost can accurately detect the effects of changes in policy. If the machine model on which the application is run changes, cost can offer an insight as to how well the program and policy runs on the given machines.

The basic concepts of the model are a machine, a trace, a placement, a policy, and the cost function. They are described in detail in the following sections.

4.1.1 Machines

A machine μ is defined by a set of processors, a set of memories, and some parameters that represent the speed of the various memory operations. The set of processors is denoted Π, the set of memories $M = \Pi$ or $M = \Pi \cup \{\mathbf{global}\}$, and the parameters $r > 1$, $g > 1$, $R > 2r$, and $G > 2g$. Each parameter is measured in units of time equal to that of a single-word local cache hit (or memory reference in uncached machines). Lower-case r is the amount of time that it takes a processor to access a word from another processor's memory.[2] Capital R is the amount of time that it takes for a processor to copy an entire block[3] from another processor's memory. If a machine has global memory (memory that is not associated with any particular processor, but rather is equidistant from all processors—this could be main memory in a cached machine or "dance hall" memory in a NUMA) then the amount of time to access a word in global memory is g, while G is the cost of moving an entire block from global memory to a local memory. The model requires that if g and G are not infinite, then $r \geq g$ and $R \leq 2G$. Otherwise, if $r < g$ it would never make sense to use the global memory; if $R > 2G$ then one could make a copy from a remote memory by copying first to global and then from there to the destination. The symbol $\sharp$ denotes the number of elements in a finite set. To eliminate trivial cases, there must be more than one processor, i.e. $p = \sharp\Pi > 1$.

The model uses the same coherence constraint as the ACE kernel: at the time a block is written, there may only be one copy of that block. The model does not consider other coherence constraints such as write-update or write-update/write-invalidate hybrids. It also assumes sequential consistency and so excludes weak consistency models.

Some systems may not have all of the features described above. The BBN Butterfly [14], for example, has memory at each processor but no caches and no global memory; it can be modeled by making G and g infinite. In a coherently cached system where it is not possible to read a single word from a line stored in a different cache, r is infinite. If blocks can only be loaded from main memory and not directly from another cache, then R is also infinite, and G is the time to access the main memory.

[2] "Another processor's memory" could be main memory associated with a particular processor in a NUMA system, or a cache line in a coherently cached machine or non-coherently cached NUMA; r may be infinite if no direct remote access is permitted.

[3] Throughout this work the word "block" is used for the unit of memory that can be moved from one location to another; this formalism applies equally well to pages and cache lines; "block" is meant to represent either, depending on context.

4.1.2 Traces

A *trace* T is a list (T_t) of references indexed by Time. The word "Time" (with a capital "T") represents the index of a particular memory reference within a trace; it is not directly related to the execution time of a program. This list is meant to capture all the memory activity of all processors, in order, over the lifetime of the program. An important simplifying assumption is that a total ordering of memory references exists, and that it is invariant, regardless of the hardware model and policy decisions.

Cost is roughly analogous to execution time. Thus, regardless of the policy or hardware considered in a particular execution, the Time of the trace is the same; it is the number of references made when the trace was generated. The Time set, τ, is a set of integers from 0 to $n-1$ where $n = \sharp\tau$, the number of references in the trace. A *reference* is a triple (a, j, w) where a is the memory address of the word being referenced, $j \in \Pi$ is the processor making the reference, and w is either **read** or **write**. If ρ is the set of all possible references, a trace $T \in \rho^{\tau}$. $\mathrm{Trc}(\mu)$ denotes the set of all traces for machine μ.

In practice, a change in policy will alter program timings, leading to a different trace, which in turn may change the behavior of the policy, and so on. At the very least a change in policy will change the interleaving of references from different processors; the approach described here ignores this. One could adjust the interleaving during trace analysis, based on per-processor accumulated costs, but this approach would run the risk of introducing interleavings forbidden by synchronization constraints in the program. It would also at best be a partial solution, since the resolution of race conditions (including "legitimate" races, such as removing jobs from a shared work queue) could lead to a different execution altogether. Forbidden interleavings could be avoided by identifying synchronization operations in a trace, and never moving references across them, but even this approach fails to address race conditions. On-the-fly trace analysis, such as performed in TRAPEDS [94], could result in better quality results, but only at a significant cost for maintaining a global notion of time (e.g. synchronizing on every simulated machine cycle). Section 4.3 describes a series of experiments designed to measure the sensitivity of the simulation results of the types used in later chapters to the sorts of changes in instruction interleaving that are likely to be present in the traces. It finds that the possible errors in interleaving have little effect on the results of the simulations.

4.1.3 Placements and Policies

A trace describes an application without specifying the location(s) within the machine at which pages reside over Time. These locations are known as a placement; they are chosen by a policy. The model considers all caches and memories to be of infinite size, so blocks are never evicted from the cache or from memory due to lack of space. The model does not consider contention; if its effects are significant, then the results will be skewed. Furthermore, since the policy choices are assumed to have no effect on the trace references, placement decisions made for different pages have no impact on one another, and so policies may treat pages independently. Therefore, the following presentation is limitied to describing a single block, without loss of generality. The overall cost for an application is the sum of costs for all of its blocks.

A *placement* P is a Time-indexed list (P_t) of location sets, where $P_t \subseteq M$, $\sharp P_t > 0$, and $(T_t.\text{type} = \textbf{write}) \Rightarrow (\sharp P_t = 1)$. That is, each placement set is non-empty, and is a singleton whenever the corresponding reference is a write: this is the embodiment of the chosen coherence constraint. A *policy*, $\mathcal{P}$, is a mapping from traces to placements. Given a machine μ the set of all policies for that machine is denoted $\text{Pol}(\mu)$.

4.1.4 Cost

The function c maps a trace and a placement for that trace into an integer, called the *cost* of the placement for the trace. The cost of a placement on a trace is the sum of two components: the cost due to references and the cost due to page movement. The reference component, c_{ref}, is defined as:

$$c_{\text{ref}}(P,T) \equiv \sum_{t=0}^{n-1} \begin{cases} 1 & \text{if } T_t.\text{proc} \in P_t \\ g & \text{if } \textbf{global} \in P_t \text{ and } T_t.\text{proc} \notin P_t \\ r & \text{otherwise} \end{cases} \tag{4.1}$$

That is, each reference to a local block costs 1; g is the cost for each reference to a page that is global memory, but not in local memory; and r is the cost for each reference that must be made to a page in some other processor's memory. The block movement component, c_{mv}, is the cost required to make the block moves described by the placement.

$$c_{\text{mv}}(P,T) \equiv \sum_{t=1}^{n-1} \begin{cases} G \cdot \sharp(P_t \setminus P_{t-1}) & \text{if } \textbf{global} \in P_{t-1} \cup P_t \\ R \cdot \sharp(P_t \setminus P_{t-1}) & \text{otherwise} \end{cases} \tag{4.2}$$

The sum here runs from 1 to $n-1$ instead of from 0 to $n-1$, because no movement cost is charged for the initial placement of the page at $t = 0$. The movement component of the cost is simply what is required to move the page into any new locations that it assumes.

Finally, then, $c(P,T) \equiv c_{\text{ref}}(P,T) + c_{\text{mv}}(P,T)$. The related function, $c_{\text{po}}(\mathcal{P},T) \equiv c(\mathcal{P}(T),T)$, maps policies and traces to cost. Since c and c_{po} are similar in meaning and should be easy to tell apart from context, henceforth, c will be used to denote both.

While it is not explicitly mentioned in the definition of c, different machines have different cost functions. So, if we are considering two machines μ and μ', their cost functions will be denoted c and c', respectively.

4.1.5 Optimality

Given a machine μ and a trace $T \in \text{Trc}(\mu)$, a placement $P \in \text{Plc}(T)$ is said to be *optimal* if $\forall Q \in \text{Plc}(T) : c(P,T) \leq c(Q,T)$. Similarly, a policy $\mathcal{P} \in \text{Pol}(\mu)$ is optimal if $\forall \mathcal{Q} \in \text{Pol}(\mu), \forall T \in \text{Trc}(\mu) : c(\mathcal{P},T) \leq c(\mathcal{Q},T)$. That is, a placement for a trace is optimal if it has cost no greater than that of any other placement for that trace; a policy for a machine is optimal if it generates an optimal placement for any trace on that machine.

A policy $\mathcal{P} \in \text{Pol}(\mu)$ is *on-line* if $\forall T, T' \in \text{Trc}(\mu), \forall i \in 0..n-1 : (T_{0..i} = T'_{0..i}) \Rightarrow (\mathcal{P}(T)_{0..i} = \mathcal{P}(T')_{0..i})$. In other words, $\mathcal{P}$ is on-line if the portion of any placement generated by $\mathcal{P}$ for Time 0 to i depends only on the references made up to and including Time i; i.e. iff $\mathcal{P}$ uses no future knowledge. A policy is *off-line* if it is not on-line. These definitions correspond to the standard notions of on- and off-line algorithms.

Proposition 1 *Given machine μ, any optimal policy $\mathcal{O} \in Pol(\mu)$ is off-line.*

PROOF: Let machine μ with processor set Π, memory set M, and parameters r, g, R and G and optimal policy $\mathcal{O} \in \text{Pol}(\mu)$ be given. Because $\sharp\Pi = p > 1$, we may choose distinct processors $p_1, p_2 \in \Pi$.

Consider trace T_1 defined to be $10R$ writes by p_1 followed by 1 write by p_2 followed by $10R$ writes from p_1. The only optimal placement P_1 for T_1 starts the page at p_1 at the beginning of the execution and leaves it there for the entire run. Consider now trace T_2 defined to be $10R$ writes by p_1 followed by $10R$ writes by p_2. The only optimal placement P_2 for T_2 starts the page at p_1 and moves it to p_2 at Time $10R$. Since $\mathcal{O}$ is optimal and P_1 and P_2 are the unique optimal placements for T_1 and T_2 respectively, $\mathcal{O}(T_1) = P_1$ and $\mathcal{O}(T_2) = P_2$. Since T_1 and T_2 are identical up to reference $10R + 1$, but yet $\mathcal{O}(T_1)$ and $\mathcal{O}(T_2)$ differ at Time $10R$, we conclude that $\mathcal{O}$ is off-line. □

The following theorem states that, as a consequence of the model and the definition of optimality, an optimal placement for a given trace on a given machine is also optimal for the same trace on a machine in which the ratios of $r-1$, R, $g-1$ and G remain constant. That is: if the local memory (cache hit) speed is changed, but the ratio of the additional cost of remote and global reference over local reference to remote and global block moves is unchanged, the placement is still optimal for the new machine. In essence, this means that local memory speed does not have an effect on the optimal placement.

This theorem is called the s-theorem, because of the scale factor s used in its statement.

Theorem 1 *Given machine $\mu = (\Pi, M, g, r, G, R)$, $s > 0$, trace T and optimal policy $\mathcal{O} \in Pol(\mu)$, $\mathcal{O}(T)$ is an optimal placement for machine $\mu' = (\Pi, M, s(g-1)+1, s(r-1)+1, sG, sR)$.*

PROOF: Let machine μ, trace T, s and $\mathcal{O}$ be given and μ' defined as in the hypothesis. Define c to be the cost function for μ and c' to be the cost function for μ'. Define P to be $\mathcal{O}(T)$. Let placement Q for T be given. Because P is optimal on μ, $c(P,T) \leq c(Q,T)$. Define λ_P to be the number of local references made by P, ρ_P the number of remote references, γ_P the number of global references, Φ_P the number of remote moves and Γ_P the number of global moves. Define λ_Q, ρ_Q, γ_Q, Φ_Q and Γ_Q similarly for placement Q. By definition of cost and because P is optimal, $c(P,T) = r\rho_P + g\gamma_P + \lambda_P + R\Phi_P + G\Gamma_P \leq r\rho_Q + g\gamma_Q + \lambda_Q + R\Phi_Q + G\Gamma_Q = c(Q,T)$. Since $\lambda_P + \rho_P + \gamma_P = \sharp T = \lambda_Q + \rho_Q + \gamma_Q$ we may subtract them from both sides of the inequality, and since $s > 0$, we may multiply without changing the sense of the inequality, giving $s(r-1)\rho_P + s(g-1)\gamma_P + sR\Phi_P + sG\Gamma_P \leq s(r-1)\rho_Q + s(g-1)\gamma_Q + sR\Phi_Q + sG\Gamma_Q$. Adding in the terms equal to $\sharp T$ subtracted above and observing the definition of c' yields $c'(P,T) \leq c'(Q,T)$. Since placement Q was arbitrary, by definition of optimality P is optimal on μ'. □

Using the s-theorem, for a given trace it is possible to directly compute the cost of an optimal placement on one machine given the cost of an optimal placement on another, providing that the machines are related as described in the premise of the theorem. The following corollary describes the computation.

Corollary 1 *Given μ, s, μ', T and $\mathcal{O}$ as in the previous theorem, and given optimal policy $\mathcal{O}'$ for μ', if n is the length of trace T, c is the cost function for μ and c' the cost function of μ', then $\frac{c'(\mathcal{O}'(T),T)}{n} = 1 + s(\frac{c(\mathcal{O}(T),T)}{n} - 1)$.*

PROOF: Define P to be the placement $\mathcal{O}(T)$ (for machine μ). Define λ, ρ, γ, Φ and Γ as in the previous proof. If c is the cost function for machine μ, then $c(P,T) = \lambda + r\rho + g\gamma + R\Phi + G\Gamma$. Since every reference made in the trace is either local, global or remote, $\lambda + \gamma + \rho = n$. Therefore, $c(P,T) = n + (r-1)\rho + (g-1)\gamma + R\Phi + G\Gamma$ and $\frac{c(P,T)}{n} = 1 + \frac{(r-1)\rho + (g-1)\gamma + R\Phi + G\Gamma}{n}$. By the previous theorem, P is optimal for T on machine μ'. Since by hypothesis $\mathcal{O}'$ is optimal, $c'(\mathcal{O}'(T),T) = c'(P,T)$. By definitions of cost and μ', $c'(P,T) = \lambda + (s(r-1)+1)\rho + (s(g-1)+1)\gamma + sR\Phi + sG\Gamma = n + s(r-1)\rho + s(g-1)\gamma + sR\Phi + sG\Gamma$. Dividing by n, we have $\frac{c'(P,T)}{n} = 1 + s\frac{(r-1)\rho + s(g-1)\gamma + sR\Phi + sG\Gamma}{n}$. Subtracting one from the final formula for $c(P,T)/n$ above and substituting yields $\frac{c'(\mathcal{O}(T),T)}{n} = \frac{c'(P,T)}{n} = 1 + s(\frac{c(P,T)}{n} - 1) = 1 + s(\frac{c(\mathcal{O}(T),T)}{n} - 1)$. □

Consider now the function that carries r into $c(\mathcal{O}, T)$ (recall that r, the remote reference time is part of the machine definition, and so is an implicit parameter in both c and $\mathcal{O}$). If r is thought of as being a real valued variable, this function is continuous, monotonically non-decreasing in r, and is piecewise linear. It is differentiable everywhere except at those points where it changes slope, and its derivative is a step function that grows as r gets smaller.

To see why this is true, consider a range of values of r over which the optimal *placement* does not change. The cost of this optimal placement over this range is some constant representing the cost of local and global references and page moves plus r times the number of remote references. This is obviously continuous and linear in r, and non-decreasing. Now consider a point at which the optimal placement changes.[4] Since $\mathcal{O}$ is optimal, any change due only to a reduction in r will result in more remote references. Therefore, on the side of the change where r is smaller, the optimal cost will have more remote references and a smaller amount of cost due to other factors; that is, the derivative of cost with respect to r has a jump discontinuity at the point where the placement changes.

Similar things are true when considering the function that carries R, G or g into the optimal cost.

4.2 Computing Optimal NUMA Placements

A placement can be thought of as a Time-ordered walk through the space of possible page replication sets. At each point in Time the fundamental question to be answered is whether to leave a page in remote or global memory, or to migrate or replicate it into global or remote memory. For some machines, one or the other of these options may not exist: there may be no global memory, remote reference may not be supported, and so on. In any case, brute-force exploration of the search space is obviously impractical: the number of possible placements is on the order of n^{2^p}.

The algorithm for computing an optimal placement is presented in two versions for the sake of clarity. The first computes a placement that is optimal under the assumption that replication is disallowed; equivalently, it treats all references as if they were writes. Then, the algorithm is extended to permit replications. Both algorithms compute the cost of an optimal placement rather than the placement itself. Since the computations are constructive, it is simple to extend them to produce the placement.

[4]In fact, at any such point there will be at least two different optimal placements with equal cost; assume $\mathcal{O}$ picks one arbitrarily.

```
for m ∈ $M$ cost_so_far[m] ← 0
for t ← 0 to $n - 1$                                  /* for all references in trace */
    cheap_cost ← cost_so_far[global]
    C ← $G$; cheapest ← global                        /* C is cost for cheapest */
    for m ∈ $(M \setminus \{global\})$
        if cost_so_far[m] + $R$ < cheap_cost + C
            cheap_cost ← cost_so_far[m]
            C ← $R$; cheapest ← m
    new_cost_so_far[$T_t$.proc] ← MIN (
        cost_so_far[$T_t$.proc] + 1,                  /* use copy already here */
        cost_so_far[cheapest] + C + 1)                /* get it now */
    new_cost_so_far[global] ← MIN (
        cost_so_far[global] + $g$,                    /* use global copy */
        cost_so_far[cheapest] + $G + g$)              /* migrate from cheapest */
    for $m \in (M \setminus \{T_t.\text{proc} \cup \text{global}\})$
        new_cost_so_far[m] ← MIN (
            cost_so_far[m] + $r$,                     /* use copy already there */
            cost_so_far[cheapest] + C + $r$)          /* migrate from cheapest */
    cost_so_far ← new_cost_so_far                     /* update whole array */
return $\text{MIN}_{m \in M}$ (cost_so_far[m])
```

Figure 4.1: Algorithm for computing optimal cost without replication

4.2.1 Computing Optimality Without Replication

The first version of the optimal algorithm (Figure 4.1) assumes that replications are prohibited. This algorithm resembles the solution to the full version of the problem, but is simpler and easier to understand. To fit it into the framework of the cost metric presented in section 4.1.4, it pretends that all references are writes.

The algorithm uses dynamic programming to determine, after each reference, the cheapest way that the page could wind up in each possible memory location. At Time t, for each memory, the cheapest way that the page could wind up there is necessarily an extension of the cheapest way to get it to some (possibly different) location at Time $t - 1$. The principal data structure, then, is an array of costs, "cost_so_far," indexed on memories $m \in M$. At Time t, cost_so_far[m] contains the cost of the cheapest placement for the trace $T_{0..t}$ that would end with the page at m. At the end of the algorithm, the cost of the cheapest overall placement is the minimum over $m \in M$ of cost_so_far[m]. The key to dynamic programming, of course, is that while the algorithm never looks back in the trace stream, it does not know where the page might be located at the Time that a reference is made.

Only at the end of the trace is the actual placement known.

The algorithm in Figure 4.1 runs in time $O(np)$. There exists another version that runs in time $O(n)$, found by Paul Dietz. It uses the observation that there is always an optimal placement that never moves a page to a processor other than the one making the current reference. Only the processor making the current reference has its cost updated; a single variable is used to keep track of the others. In that way, each reference is processed in constant time, regardless of the number of processors.

4.2.2 Incorporating Replication

The obvious extension for the general case (with replication) is simply to enlarge the set M to include all possible replication states, and to enforce coherence by assuming that the transitions into non-singleton states are of infinite cost when the reference is a write. Unfortunately, this extension increases the time complexity of the inner loops of the algorithm from $O(p)$ to $O(2^p)$ for the cases where the reference is a read. This is a severe penalty even when using the eight processor traces generated on the ACE as described in Chapter 2; for large machines it is out of the question.

Fortunately, it is not necessary to search this large state space. Name the Time interval between two writes with at least one read and no other writes between them a *read-run.* Because of the coherence constraint, at the beginning and end of a read-run the page state must be a singleton. There is no cost benefit in removing a copy of the page inside of a read-run, so we can ignore all such placements. Similarly, if the page will be replicated to a memory inside of the read-run, there is no cost penalty involved in making the replication on the first reference of the read-run. So, for any given read-run, all that needs to be decided is the set of processors to which to replicate; there exists an optimal placement that replicates to these processors at the beginning of the read-run and destroys the replicates on the terminal write, without changing the replication state in between. Furthermore, the set of processors to which to replicate during a given read-run depends only on the processor(s) making the beginning and terminal writes, the location of the page at the beginning and end of the read run, and the number of reads made by each processor during the run.

Armed with these observations, the algorithm in Figure 4.1 can be extended to the general case. The new version appears in Figure 4.3. The function in Figure 4.2 computes the cost of a read-run, given the starting location, the replication set and the number of reads made by each processor during the run. For the sake of simplicity, this function assumes that there is no global memory. The modifications required to handle global memory are straightforward but tedious.

```
FUNCTION read_run_cost (start : location; rep_set : set of location;
    reads_from : associative array [processor] of cost) : cost
running_total ← 0
for each j ∈ domain (reads_from)
    if j ∈ rep_set
        running_total + ← reads_made[j]
    else
        running_total + ← $r$ * reads_made[j]
    if start ∈ rep_set
        running_total + ← $R$ * (♯rep_set − 1)
    else
        running_total + ← $R$ * ♯rep_set
return (cost_so_far[start] + running_total)          /* cost_so_far is global */
```

Figure 4.2: Function to compute the cost of a read-run, no global memory

The new algorithm still uses dynamic programming, but while the state space was updated on every reference in the old version, it is only updated on writes in the new one. The space that is tracked remains M. In addition, while formerly at each step we had to consider the possibilities of starting the page at the current location, or in the cheapest location among the rest of the processors, we must now also consider the possibility that a processor may effectively become the cheapest by virtue of a savings in references during the preceding read-run, even if these references would not otherwise justify replication.

4.3 Validation of the Trace Analysis Technique

The ACE tracer slows down the execution of a program by between two and three orders of magnitude. This has some effect on the order in which references are made. While all processors are slowed more-or-less uniformly, the *dilation* effect [62] will overwhelm any difference in execution times of the various machine instructions. On the ACE's processor, most instructions take only 1 cycle to execute. The notable exceptions are memory reference instructions and floating point operations, which take somewhat more time depending on the instruction, on whether the memory is busy, etc. Koldinger *et al.* [62] investigated these sorts of effects in the related area of coherent cache simulation, and found the performance differences due to dilation to be negligible. Since the optimal policy guarantees small changes in cost in response to small changes in the trace input (it is, in some sense, continuous in the trace, as it is in the machine speed parameters),

```
refs_to_pay_for_repl ← $R/(r-1)$
for j ∈ Π cost_so_far[j] ← 0
reads_from ← empty                                    /* associative array */
for t ← 0 to $n-1$                                    /* for all references in trace */
    if $T_t$.type = **read**
        if $T_t$.proc ∈ domain (reads_from)
            reads_from[$T_t$.proc] + ← 1
        else
            reads_from[$T_t$.proc] ← 1
    else /* write */
        repl_procs ← {j ∈ domain (reads_from) *such that* reads_from[j] > refs_to_pay_for_repl}
        cheapest ← j ∈ $M$ *such that* cost_so_far[j] *is least*
        min_nonrep_proc ← j ∈ (Π \ repl_procs)
            *such that* cost_so_far[j] $-(r-1)$ * reads_from[j] *is least*
            /* if repl_procs = Π, pick an arbitrary processor */
        for j ∈ Π
            /* We follow one of three possible replication patterns: start where we finish,
               start at the place that was cheapest to begin with, or start at the place that
               was cheapest but not in the set of memories for which the number of reads
               was enough to offset the cost of replication by itself. */
            new_cost_so_far[j] ← MIN (
                read_run_cost (j, {j} ∪ repl_procs, reads_from),
                read_run_cost (cheapest, {cheapest, j} ∪ repl_procs, reads_from),
                read_run_cost (min_nonrep_proc, {min_nonrep_proc, j} ∪ repl_procs, reads_from))
            if $T_t$.proc = j /* write by ending processor */
                new_cost_so_far[j] + ← 1
            else /* write by another processor */
                new_cost_so_far[j] + ← r
        cost_so_far ← new_cost_so_far                 /* update whole array */
        reads_from ← empty
/* The entire trace has been processed. Clean up if we're in a read-run. */
if $T_{n-1}$.type = **write**
    return $\text{MIN}_{j \in \Pi}$ (cost_so_far[j])
repl_procs ← {j ∈ domain (reads_from) *such that* reads_from[j] > refs_to_pay_for_repl}
cheapest ← j ∈ $M$ *such that* cost_so_far[j] *is least*
min_nonrep_proc ← j ∈ (Π \ repl_procs)
    *such that* cost_so_far[j] $-(r-1)$ * reads_from[j] *is least*
    /* if repl_procs = Π, pick an arbitrary processor */
for j ∈ Π
    new_cost_so_far[j] ← MIN (
        read_run_cost (j, {j} ∪ repl_procs, reads_from),
        read_run_cost (cheapest, {cheapest, j} ∪ repl_procs, reads_from),
        read_run_cost (min_nonrep_proc, {min_nonrep_proc, j} ∪ repl_procs, reads_from))
return $\text{MIN}_{j \in \Pi}$ (new_cost_so_far[j])
```

Figure 4.3: Optimal policy computation, no global memory

it is natural to expect its performance to be even less affected by dilation.

As noted in section 4.1.2, a more fundamental problem with the evaluation of multiprocessor memory systems based on static trace interleavings is a failure to capture the influence of the simulated system on the references that "should" have occurred. In the system described here, this feedback should appear in two forms: fine-grain changes in instruction interleaving, and coarse-grain "reference gaps" in the activity of individual processors. Instruction timings depend on whether the operands of loads and stores are local or remote. If two policies place a page in a different location at different points in time, then instructions will execute faster on some processor(s) and slower on others, and the interleaving of instructions from different processors will change. Similarly, when a policy decides to move a page, the processor performing the move will stop executing its user program until the move is complete. Since this could potentially take a long time (particularly in a system with large pages and/or large interprocessor latencies), other processes might make a large number of references in the interim. Since the times at which the page moves would occur are not known when the applications are traced, and in general depend on the parameters of the simulation later performed on the trace, no such gaps appear in the traces.

The performance effect of errors in the trace depends not only on the errors themselves, but also on the machine on which the trace is being simulated. While many different machine models are considered here, four representatives are used in the validation study. The ACE model has fast global memory and no special block transfer hardware. The Butterfly has no global memory, a fast block transfer and a 15–1 remote to local access time ratio. The NUMA machine has a 100–1 remote to local ratio, and CC (coherently cached) has 64 byte pages and no remote reference at all. The ACE and Butterfly are described in detail in Chapter 5, while NUMA and CC are described in Chapter 6.

To evaluate the impact of changes in fine-grain instruction interleavings (which might be caused by contention for the shared buffer in the tracer), I wrote a filter program that reorders individual references in a trace, with a probability that is high for nearby references, and drops off sharply for larger Time spans. Specifically, the filter keeps a buffer of 100 references from the incoming trace stream. Initially, this buffer is filled with the first 100 references. The filter then randomly chooses an entry from the buffer, emits the oldest buffered reference made by the processor whose entry was selected, and reads a new reference to replace it. After running the traces through this filter, the largest measured degree of change in optimal performance among all applications for the ACE machine model was .007%. For the Butterfly model it was .01% and for NUMA it was .8%.

Unlike the other machine models, some of the applications showed a large performance difference in CC, with `cholesky` the largest at 11%. This size deviation is disturbing, and also striking in that it is out of line with the results

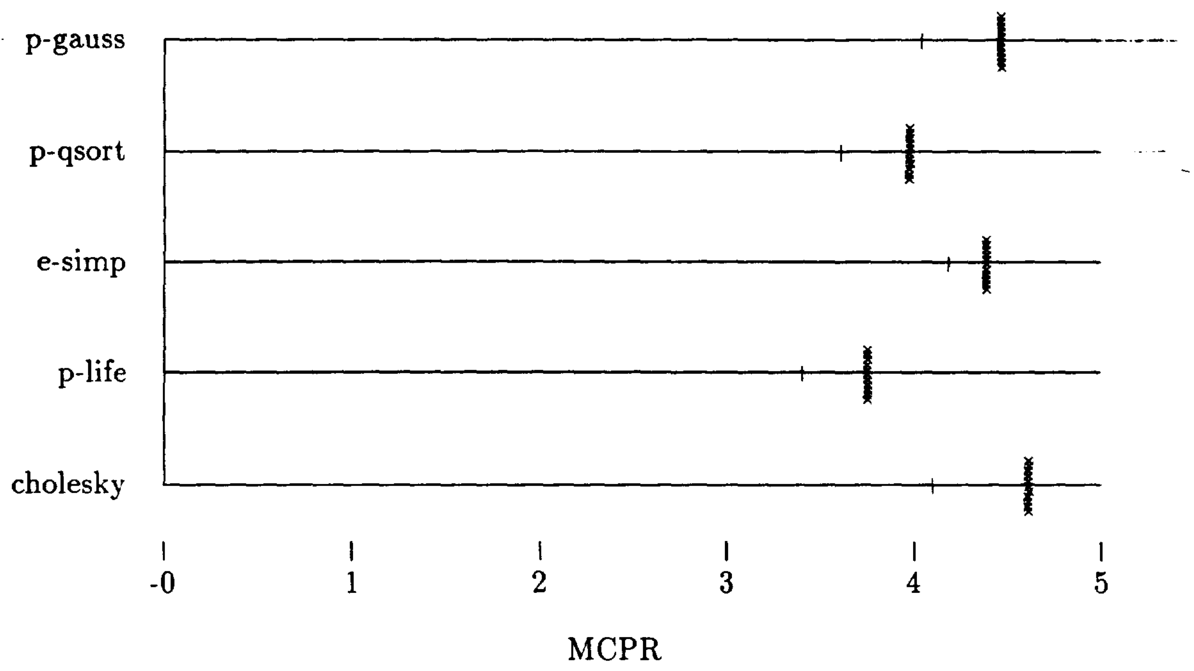

Figure 4.4: Unmodified vs. local perturbations for CC model

for the rest of the applications and machine models. The applications with the largest differences between optimal and locally-perturbed optimal are `cholesky`, `p-gauss`, `p-qsort` and `p-life`. All of these applications share memory at a fairly fine grain. The local perturbation does not take into account the synchronization behavior of the programs, and so could move references over a synchronization point; in the real tracer, only one reference by a given processor could be moved over a synchronization point. If there are many sync points, and the perturber does indeed move references over them frequently, then the level of sharing in the program would increase and the cost would go up, particularly in the CC model where remote reference is unavailable. Furthermore, most perturbations of this type would move roughly the same number of references across sync points, so if a number of different perturbed runs (using different random number seeds) were compared, they would all have similar cost; that cost would be significantly higher than that of the unperturbed run.

Figure 4.4 shows the results of running multiple local perturbation runs for each of the applications that had a large difference in optimal performance for the CC model. Each horizontal line in the graph represents one application, and is labeled appropriately at its left hand side. On this line, a single vertical tick mark shows the MCPR cost of the unperturbed trace. An "x" is used to represent a run in which the local perturbation was introduced. While on these graphs it

App.	Local				Gap			
	ACE	Bfly	NUMA	CC	ACE	Bfly	NUMA	CC
`e-fft`	.002%	.012%	.02%	.4%	0	.18%	1%	.3%
`e-simp`	.001%	.001%	.03%	5%	.001%	.039%	.7%	.02%
`e-hyd`	0	0	.1%	1%	0	.001%	1%	3.5%
`e-nasap`	.007%	.03%	.09%	.8%	.004%	.15%	1.5%	.23%
`gauss`	0	0	.003%	.002%	.02%	.07%	.4%	.09%
`chip`	0	0	.05%	.4%	0	0	.5%	.3%
`bsort`	0	0	0	0	0	0	.003%	.001%
`kmerge`	0	0	.007%	.01%	0	0	.01%	.004%
`plytrace`	0	0	.02%	.4%	.01%	.06%	3%	.6%
`sorbyc`	0	.01%	.001%	.04%	0	.01%	.02%	.3%
`sorbyr`	0	.007%	.01%	.01%	0	.07%	1.2%	.09%
`matmult`	0	.03%	.3%	.9%	0	.3%	.2%	1%
`mp3d`	0	0	.8%	2.1%	0	.002%	1.1%	1.5%
`cholesky`	0	.004%	.08%	11%	0	.03%	2%	2%
`water`	0	0	.007%	.001%	.01%	.006%	.04%	.02%
`p-gauss`	0	0	.05%	9.9%	0	.24%	1.6%	.75%
`p-qsort`	0	.01%	.2%	10%	0	.24%	1.3%	.4%
`p-matmult`	0	.01%	.08%	1.5%	0	.04%	1.3%	.4%
`p-life`	0	.005%	.4%	9.6%	.002%	.34%	2%	.4%

Table 4.1: Percentage optimal performance change due to local and gap perturbations

appears that all of the x's are at the same MCPR, in fact there are differences that are too slight to show. These results support the hypothesis that the large (5-10%) changes between unmodified and locally perturbed optimal placement costs present in some of the applications in the CC machine are an artifact of the perturbation process, and not an indication of the size of the real perturbation effect caused by the tracer.

In any case, even 20% errors in optimal performance are very much smaller than the size of effects seen in Chapter 6, and so would not significantly affect the conclusions drawn there.

To evaluate the impact of reference gaps, I wrote a second filter that randomly introduces such gaps, and again re-ran the optimal policy. The filter operates by reading the unmodified trace, and with probability one in 30,000 introduces a "gap" on one processor for 4000 references. A gap is introduced by taking any references made by the chosen processor and placing them in a queue. Once the gap has ended, as long as there are saved references, one of them will be emitted instead of a fresh reference with probability 2/3. The values 30,000 and 4000 were selected arbitrarily, but were chosen conservatively in the sense that page moves typically do not occur as often as every 30,000 references, and 4000 references is somewhat large for the amount of time for a page move. The 2/3 frequency is completely arbitrary. This filter induced performance changes up to .06% in the ACE, 0.34% in the Butterfly, 2% in the NUMA and 3.5% in the CC model.

Table 4.1 displays the differences between filtered and unfiltered results for both filters and ACE and Butterfly models as a percentage of the total cost. Differences are absolute values; sometimes the filtered values were smaller, sometimes they were larger. Values less than 0.001% are reported as 0.

These perturbations produced somewhat larger changes in performance in on-line policies, but they will still small relative to the size of the changes due to reducing the block size shown in Chapter 6.

4.4 Discussion

Implementable kernel-level policies that replicate and migrate pages suffer from the weakness that when their policies don't result in all local references with very few moves, they are unable to determine if it is because of sharing intrinsic to the application, or mistakes on the part of the policy under consideration. None of the work is easily extensible to other architectures, nor is it possible to isolate the causes of various effects in a running system.

Optimal analysis addresses these limitations. Chapter 5 shows the dependence of program performance on two basic NUMA hardware parameters: the relative cost of a block transfer (as compared to a series of individual remote accesses), and the size of a block. It also compares the performance achieved by several implementable policies with that of the optimal policy, and demonstrates how the placement decisions made by the optimal policy can be used to guide the design of an appropriate on-line policy for a given hardware architecture.

In addition to aiding the evaluation of the quality of NUMA policies, off-line optimal analysis allows evaluation of a relatively large range of shared memory multiprocessor architectures without biasing the results by selecting a particular data-movement policy. Reasoning in a formal model forces exposure of assumptions, and allows proof of theorems within that framework. All of these things provide a degree of rigor difficult to achieve in implementations.

The techniques presented in this chapter are used in subsequent chapters to examine the fundamental architectural questions: can software controlled coherence (*i.e.*, NUMA) be competitive in performance with hardware controlled coherence? If so, how should NUMA machines be built? Optimal analysis is also used to answer other questions, such as whether expensive, software implemented remote reference should be added to distributed shared memory systems, and how valuable hardware remote reference would be if it were added to cache coherent machines.

It is also possible that off-line optimal analysis could fruitfully be employed in problem domains other than multiprocessor memory management. One might, for example, create a tractable algorithm for optimizing allocation of variables to registers in a compiler, given the references to the variables that are eventually made by a particular program (that is, a trace). It would then be possible not only to measure the performance of a compiler's register allocator, but also to determine the performance inherent in different register set designs (different numbers of registers, different registers for floating point, addresses and integers vs. general purpose registers, different sizes of register windows, etc.) without having to worry that effects are due to a particular compiler, and without having to worry about implementing register allocation schemes for all of the hardware variants.

5 NUMA Policies and Their Relation to Memory Architecture

Chapter 4 describes a trace-based model of program behavior, and a tractable way to find an optimal placement for a given application on a given architecture. However, on any real machine an on-line (and consequently non-optimal) policy will be needed, and Chapter 4 offers little insight as to how such a policy should be designed.

This chapter describes simulations of on-line policies using the cost model from Chapter 4 on architectures like the ACE and Butterfly. Unlike the previous work, however, it compares the results of on-line policies with optimal performance for the same trace and machine model. In comparison to implementation-based experiments, this approach has two principal advantages:

1. The optimal algorithm gives a tight lower bound on the cost savings that could be achieved by any placement policy. Its results can be used to quantify the differences between policies, and assess the extent to which NUMA management contributes to overall program performance.
2. Watching optimal behavior as architectural parameters are changed shows the extent to which policies should be tuned to the architecture on which they are running. It also can show the usefulness of novel architectural features. As might be expected, the result of this analysis is that different architectures require different policies.

Off-line analysis also allows examination of program executions at a very fine level of detail. This examination reveals that program design, and memory usage patterns in particular, can have a profound effect on performance and on the choice of a good NUMA policy. Obtaining the best performance from a NUMA machine will require that compilation tools be aware of memory sharing patterns in the programs they are producing, that programmers work to produce "good" sharing, or, more likely, some combination of the two.

The design space of policies is large [64], but experience suggests that simple policies can work well, and that minor modifications are likely to yield only minor variations in performance. Major changes in policy are likely to be justified only by changes in architecture. Rather than search for the perfect policy on any particular machine, this chapter investigates the way in which architectural parameters determine the strategy that must be adopted by any reasonable policy.

The methodology used is that presented in Chapters 2 and 4. Section 5.2 presents the basic results. The execution cost is shown for each of the applications and policies, and then compared with the optimal policy. The ACE and PLATINUM policies are examined in detail, and the results show that they are each appropriate for the machine for which they were designed. An extension to the ACE policy that requires a small amount of additional hardware support is shown to reduce inappropriate page moves, and to improve performance over the unmodified ACE policy. Section 5.3 focuses on the tradeoff between block transfer time and the latency of remote memory references. An application is studied in detail to identify the points at which it is able to exploit faster page moves.

5.1 Implementable (Non-Optimal) Policies

In addition to the optimal policy, this chapter describes three implementable alternatives. Two of them have been used in real systems and are described in prior papers: the ACE policy [24] (see Chapter 3) and the PLATINUM policy [39]. The third policy, "Delay", is based on the ACE policy, and exploits simple hypothetical hardware to reduce the number of pages moved or frozen incorrectly.

The ACE policy can be characterized as a dynamic technique to discover a good static placement. The expectation is that the chosen placement will usually be nearly as good as a user-specified placement, and often better, and will be found with minimal overhead. The ACE policy was designed for a machine that has fast global memory ($g = 2$) and no mechanism to move a page faster than a simple copy loop ($G = 2 * pagesize + 200$ (200 is fault overhead)). When possible, pages are replicated to each processor reading them. If a page is written by a processor that has no local copy, or if multiple copies exist, then a local copy is made and all others are invalidated. After a fixed, small number of invalidations, the page is permanently frozen in global memory. The value used here (and in the ACE implementation) is four.

The PLATINUM policy was designed for a machine with no global memory, slow remote memory ($r = 15$), and a comparatively fast block transfer ($R = 3 * pagesize + 200$). Its principal difference from the ACE policy is that it continues to attempt to adapt to changing reference patterns by periodically reconsidering its placement decisions. PLATINUM replicates and moves pages as the ACE algorithm does, using an extension of a directory-based coherent cache protocol with selective invalidation [31]. The extension freezes a page at its current location when it has been invalidated by one processor and then referenced by another within a certain amount of time, t_1. Once every t_2 units of time, a daemon defrosts all previously frozen pages.[1] On the Butterfly, t_1 and t_2 were chosen to be $10ms$ and $1s$ respectively. Since time is not explicitly represented in the model, t_1 and t_2 are represented in terms of numbers of references processed. The specific values are obtained from the mean memory reference rate on an application-by-application basis, by dividing the number of references into the (wall clock) run time of the program and multiplying by $10ms$ and $1s$ respectively. The PLATINUM algorithm was designed to use remote rather than global memory, but could use global memory to hold its frozen pages.

Because they are driven by page faults, the ACE and PLATINUM policies must decide whether to move or freeze a page at the time of its first (recent) reference from a new location. Traces allow us to study the pattern of subsequent references, and confirm that the number of references following a page fault sometimes fails to justify the page move or freeze decision. Bad decisions are common in some traces, and can be quite expensive. An incorrect page move is costly on a machine (like the ACE) that lacks a fast block transfer. An incorrect page freeze is likewise costly under the ACE policy, because pages are never defrosted. A simple hardware mechanism that would allow the accumulation of some reasonable number of (recent) references from a new location before making a placement decision would be sufficient to correct for this problem.

This mechanism could be implemented by modifying the TLB by adding a counter that is decremented on each access, and that produces a fault when it reaches zero. When first accessed from a new location, a page would be mapped remotely, and its counter initialized to n. A page placement decision would be made only in the case of a subsequent zero-counter fault. This counter is similar to the one proposed by Black and Sleator [23] for handling read-only pages, but it never needs to be inspected or modified remotely, and requires only a few bits per page table entry. The value used for n here is 100. Results show that a delay of 100 is more than is normally needed, but the marginal cost of a few remote references as compared to the benefit of preventing unnecessary moves seems to justify it on machines with relatively fast remote reference, like the ACE and Butterfly. If the machine being studied had more expensive remote references, like the machines considered in Chapter 6, n would need to be reduced.

[1]The defrost behavior used here is not that of the final version of PLATINUM as described in [38]. The final version of PLATINUM uses per-page exponential backoff of defrosting for pages that are rapidly refrozen, so as to reduce the bouncing of fine grain shared pages.

5.2 Experiments

This analysis of traces attempts to answer the following questions within the formal framework of the cost model:

- To what extent can one hope to improve the performance of multiprocessor applications with kernel-based NUMA management—is the NUMA problem important?
- How closely do simple, easily implemented policies approach the performance limit of the optimal off-line policy?
- How does the choice of application programming system and style affect the effectiveness of each of the policies?
- To what extent does the effectiveness of policies vary with changes in memory architecture? Can the "strategy" used by the optimal policy be characterized as a function of these parameters?

Section 5.2.1 considers the first three questions. The final question is deferred until section 5.3.

5.2.1 Performance of the Various Policies

The performance of each of the policies on each of the applications, expressed as Mean Cost Per Reference (MCPR), appears in Figures 5.1 and 5.2–5.3, for architectures resembling the ACE and the Butterfly, respectively. Each application has a group of four bars, which represent the performance of Optimal, ACE, Delay and PLATINUM, from top to bottom. To place the sizes of the bars in context, recall that an MCPR of 1 would result if every memory reference were local and no page moves were required. For ACE hardware parameters, an MCPR of 2 is trivially achievable by placing all shared data in global memory: any policy that does worse than this is wasting time on page moves or remote references

The Importance of the NUMA Problem

For the NUMA problem to be of importance, several things must be true: memory access time must be a significant fraction of the execution time of a program; the performance difference between executions with correctly and incorrectly placed pages must be large; there must be some reasonably good solution to the problem. The results in Section 3.2.1 estimate the memory times for programs running on an ACE to be in the 25%–60% range. Newer, more aggressive processor architectures

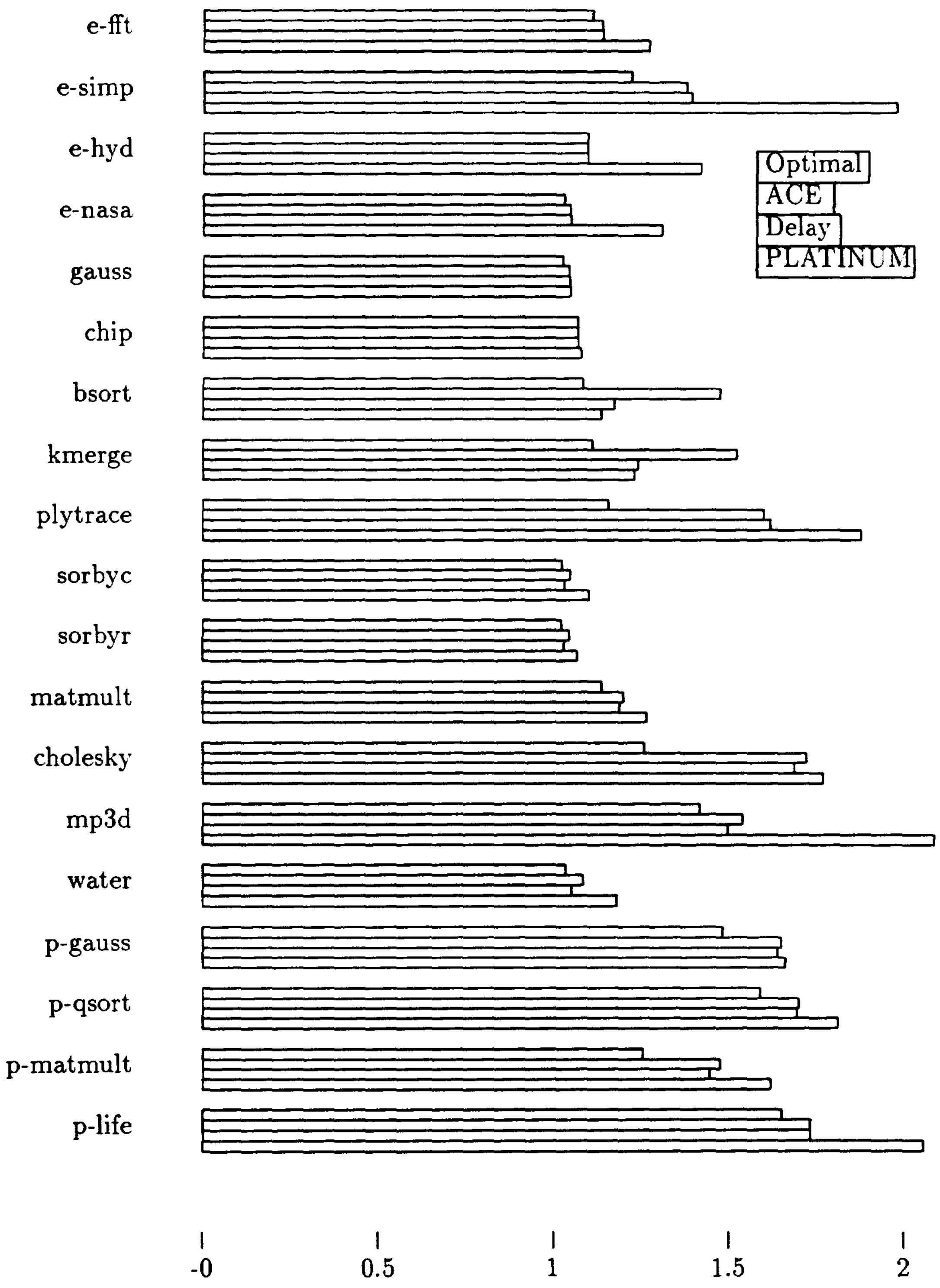

Figure 5.1: MCPR for ACE hardware parameters

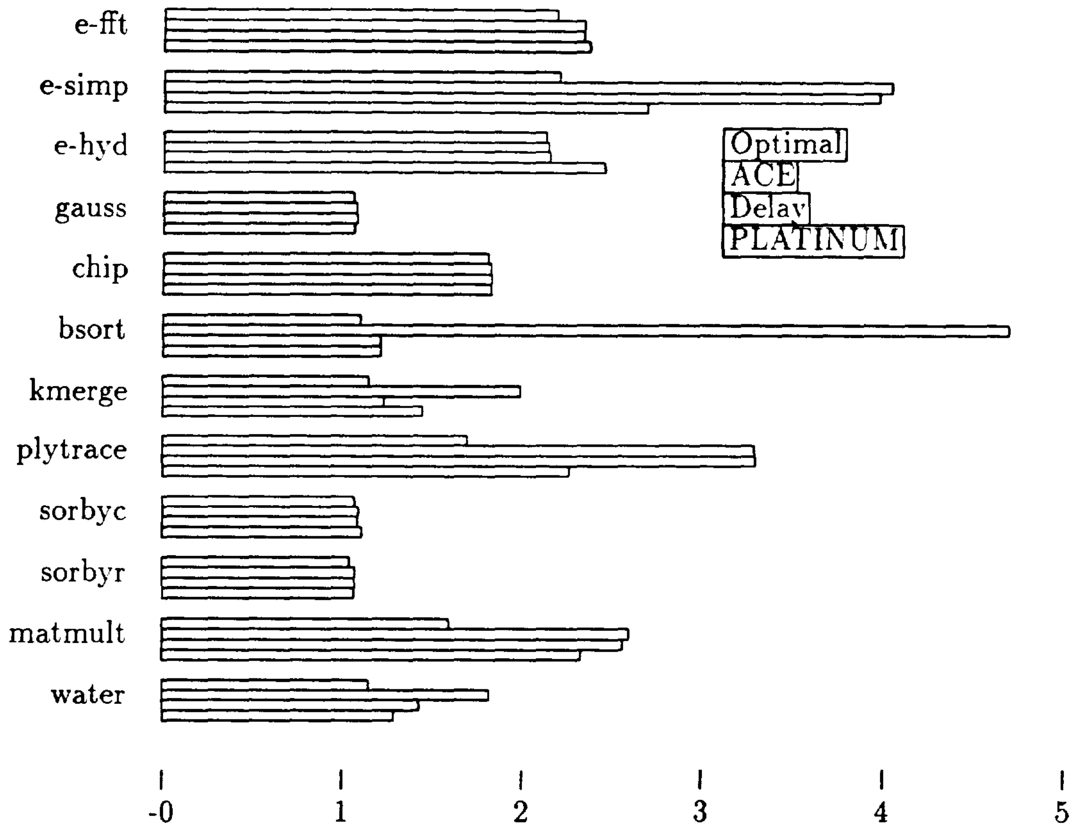

Figure 5.2: MCPR for Butterfly hardware parameters

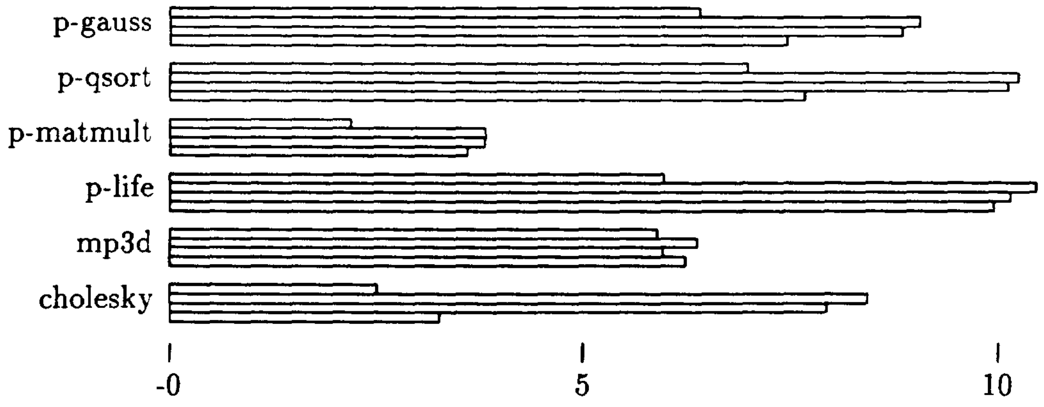

Figure 5.3: MCPR for Butterfly hardware parameters, high cost applications

will only increase this percentage, as demonstrated by the increasing emphasis on cached memory systems. [69, 78]

One possibility for NUMA management is to statically place all private data and to leave shared data in global memory or in an arbitrary local memory. This strategy will work well for applications such as `e-fft`, that have only private and fine-grained shared data, but it will not work well for others.[2] Many programs require data to migrate, particularly when remote references are costly. Examples include matrix rows lying at the boundaries between processor bands in `sorbyr`, and dynamically-allocated scene information in `plytrace`. This is demonstrated by the number of page moves performed by the optimal policy, presented in Figure 5.6. It explains why the PLATINUM policy (which is more aggressive about moving pages) generally does better than the ACE or Delay policies on a machine such as the Butterfly, in which a page move can be justified to avoid a relatively small number of remote references. The ACE policy would be wholly inappropriate on machines like those used in Chapter 6, in which the ratio of R to r is very small.

Even on a machine like the ACE, in which frozen pages are only twice as expensive to access as local pages, there is a large benefit in correctly placing pages. For all but the Presto applications and `mp3d`, an optimal placement results in an MCPR below 1.26 on the ACE (as compared to 2 for static global placement) and 2.5 on the Butterfly (as compared to 14–15 for random placement). For a program that spends half of its time accessing data memory, compared to naive placement and assuming no contention, these MCPR values translate to about a one quarter improvement in running time on the ACE, and a one half improvement on the Butterfly, As shown in the following section, the implementable policies achieve a substantial portion of this savings.

The Success of Simple Policies

Both the ACE and Delay policies do well on the ACE. The MCPR for Delay is within 15% of optimal on all applications other than `plytrace` and `cholesky`. The ACE policy similarly performs well for applications other than `plytrace`, `bsort`, `kmerge`, and `cholesky`. These programs all display modest performance improvements when some of their pages migrate periodically, and the ACE and Delay policies severely limit the extent to which this migration takes place.

All of the policies keep the MCPR below 4 for the non-Presto, non-SPLASH applications on the Butterfly, with the exception of ACE on `bsort`, and that case could be corrected by increasing the number of invalidations allowed before

[2]As will be demonstrated in Chapters 6 and 7, `e-fft` only looks as if it has fine-grained shared data because of the large page size imposed by the ACE and Butterfly models; in fact, it is false sharing that causes this illusion.

freezing. For all applications other than the Presto and SPLASH ones, PLATINUM stays near or below 2.5. This is quite good, considering that a random static placement would yield a number close to 14. The ACE and Delay policies perform slightly better than PLATINUM on applications that have only fine-grained shared and private data (`e-hyd` and `e-fft`), but the cost of moving pages that should be frozen is low enough on the Butterfly that the difference between the policies in these cases is small.

The difference between the ACE and Delay policies displays a bimodal distribution. In most cases the difference is small, but in a few cases (`bsort`, `kmerge` and `water`) the difference is large. In essence, the additional hardware required by Delay serves to prevent mistakes.

The Importance of Programming Style

The Presto applications have much higher MCPRs for both architectures, in both the on-line and optimal policies. This disappointing performance reflects the fact that these programs were not designed to work well on a NUMA machine. They have private memory but do not make much use of it, and their shared memory shows little processor locality. The shared pages in the EPEX `e-fft` and `e-hyd` programs similarly show little processor locality, but because these programs make more use of private memory, they still perform quite well.

The programs that were written with NUMA architectures in mind do much better. Compared to the Presto programs they increase the processor locality of memory usage, are careful about which objects are co-located on pages with which other objects, and limit the number of threads to the number of processors available. It is not yet clear what fraction of problems can be coded in a "NUMAticized" style.

Chapter 6 demonstrates that by reducing the page size, performance is greatly improved for most applications. This indicates that the main contributor to poor performance is false sharing, rather than real data communication. This seems to indicate that much is to be gained by altering the data-layout patterns of programs, particularly on architectures with greater remote/local ratios than those of the Butterfly.

The Impact of Memory Architecture

From the discussions above it is clear that the difference in architecture between the ACE and Butterfly machines mandates a difference in NUMA policy. It pays to be aggressive about page moves on the Butterfly. Aggressiveness buys a lot for applications such as `plytrace` and `e-simp`, that need to move some pages dynamically, and doesn't cost much for applications such as `e-fft`, which do not. At the same time, aggressiveness is a bad idea on the ACE, as witnessed by the poor performance of the PLATINUM policy on many applications (**sorbyc**, **e-simp**, **matmult**, **e-fft**, **p-gauss**). In as much as the ACE and Butterfly represent only two points in a large space of possible NUMA machines, it seems wise to explore the tradeoffs between architecture and policy in more detail. This exploration forms the subject of the following section.

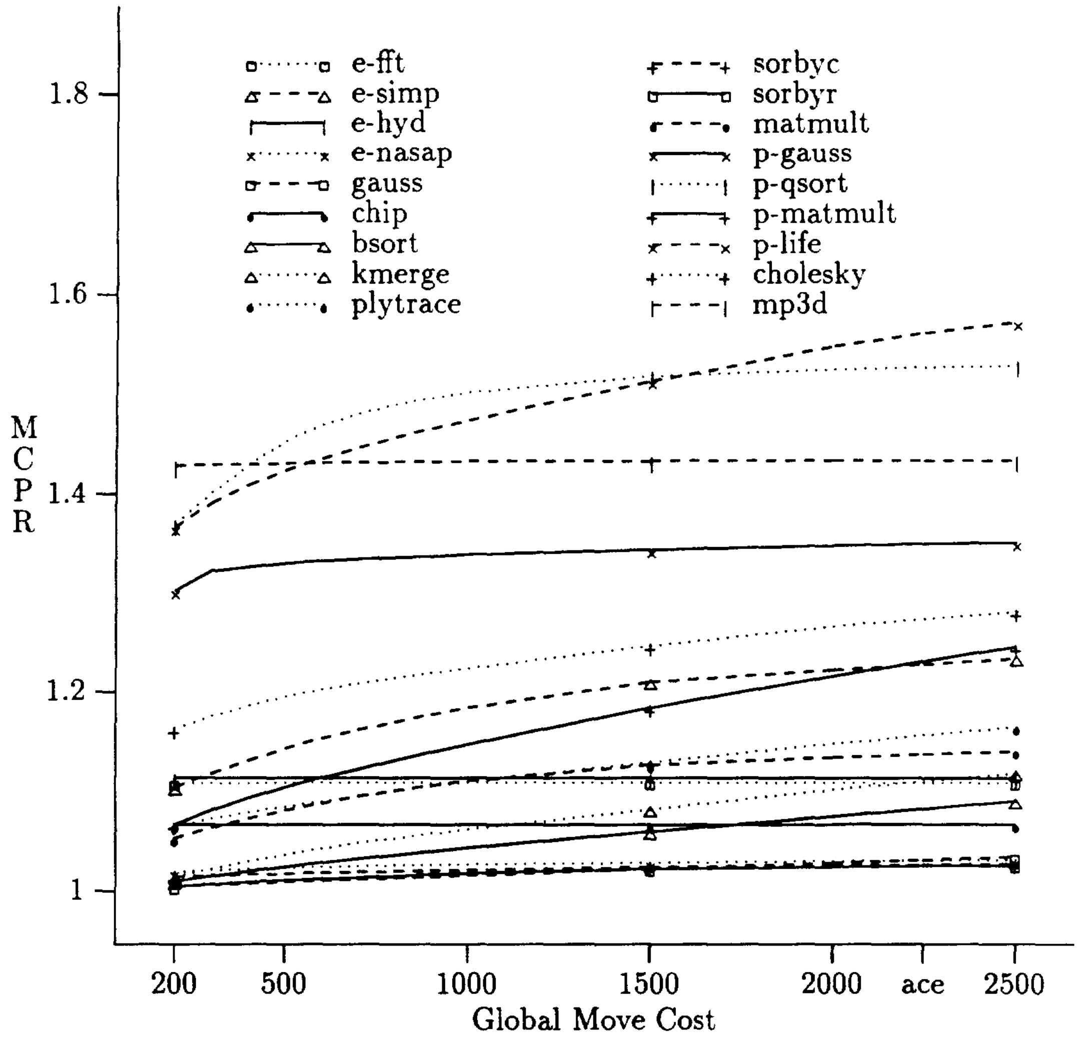

Figure 5.4: MCPR vs. G for optimal, g=2, r=5

5.3 Variation of Architectural Parameters

Figures 5.4 and 5.5 show how the performance of the optimal policy varies with the cost of a page move (G or R), for remote and global access times comparable

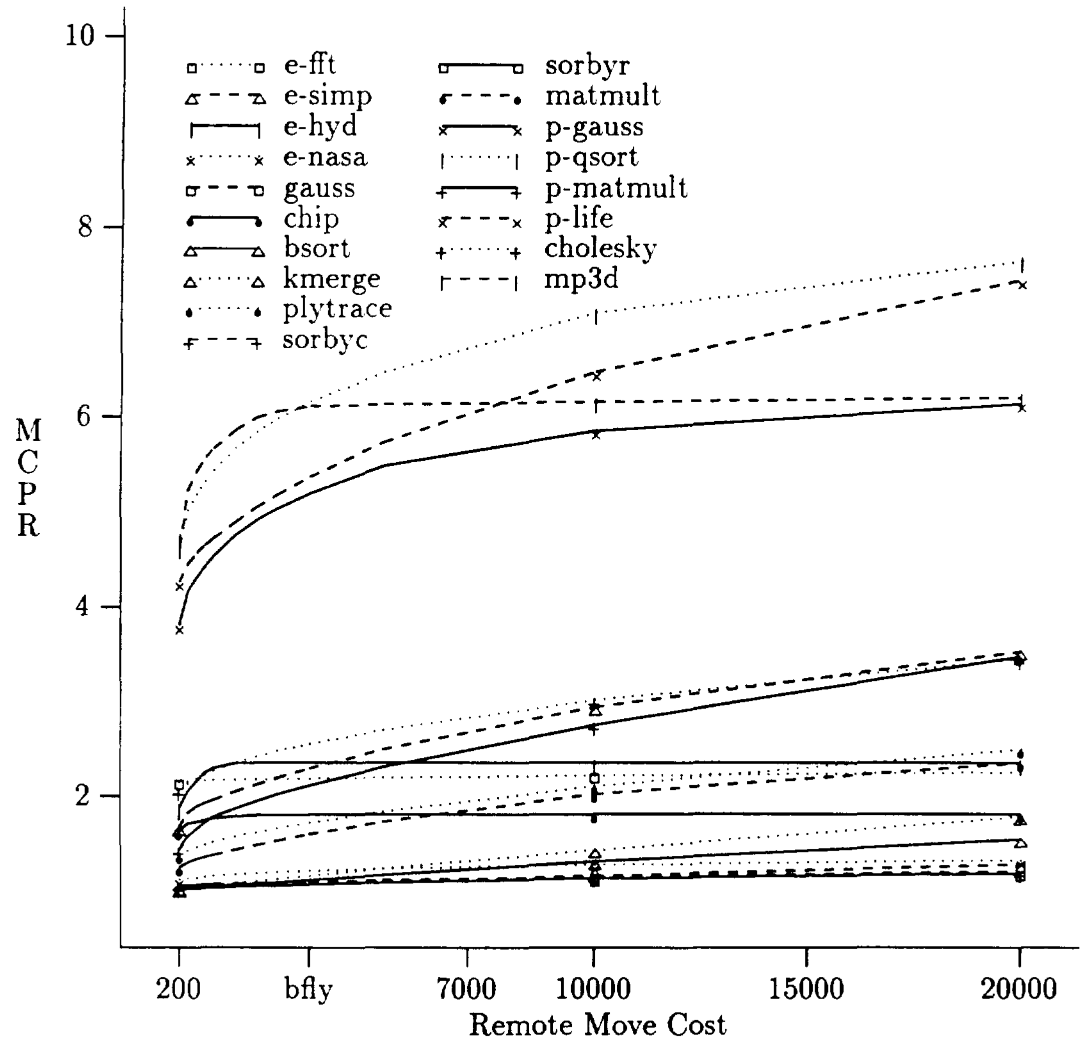

Figure 5.5: MCPR vs. R for optimal, no global, r=15

to those of the ACE and the Butterfly, respectively.

The minimum page move time represented on each graph is 200, which is assumed to be a lower bound on the time required to process a fault and initiate a page move in the kernel. 200 therefore corresponds to an infinite bandwidth, zero latency hardware block transfer. The maximum page move times on the graphs are the page size times g or r, plus a more generous amount of overhead, corresponding to a less tightly coded kernel.

As described in section 4.1.5, if R is considered to be a real-valued variable, then the cost of the optimal policy on a trace is a continuous, piecewise linear function of R. Furthermore, its slope is the number of page moves the optimal policy makes, which in turn is a monotonically non-increasing step function in R. Similar functions exist for G, g, and r, except that their slopes represent global page moves, global references, and remote references respectively. An important implication of continuity is that, given optimal placement, there is no point at which a small improvement in the speed of the memory architecture produces a disproportionately large jump in performance.

At a G or R of 0, page moves would be free. The optimal strategy would move any page on a non-local reference. This means that for a G or R of 0 the optimal MCPR of any application must be 1, regardless of the values of g and r. Since the optimal cost is continuous, the curve for every application must fall off as G or R approaches 0. This means that all the curves in Figures 5.4 and 5.5 go smoothly to 1 below their left ends. For applications such as `e-fft` that don't show much benefit from G and R down to 200, this drop is very steep.

Though different machines require different policies, any given policy[3] will be oblivious to the speed of memory operations. The curve for a given policy will therefore be a straight line on a graph like Figure 5.4, and will lie on or above the optimal curve at all points (see Figure 5.7 for an example). Because the optimal curve is concave down, no straight line can follow it closely across its entire range. This means that no single real policy will perform well over the whole range of architectures. Thus, to obtain best performance over a range of page move speeds in Figures 5.4 and 5.5 in which the optimal line curves, one must change the real policies used accordingly. However, for the applications whose curves are largely flat lines, the same policy works over the entire range.

One can plot MCPR, g (or r), and G (or R) on orthogonal axes to obtain multi-dimensional surfaces. Figures 5.4 and 5.5 show two-dimensional cuts through these surfaces. They are interesting cuts in the sense that one can imagine spending extra money on a machine to increase the speed of block transfer relative to fixed memory reference costs. It makes less sense to talk about varying the memory reference costs while keeping the block transfer speed fixed. Moreover,

[3]Here, "policy" is used in the strictest sense of the word. Changing t_1 or t_2 in PLATINUM would thus yield a new policy.

figures 5.4 and 5.5 capture all of the structure of the surfaces, at least in terms of the relationship between page move cost and memory reference cost, as shown by Theorem 1 in section 4.1.5. As a consequence of this theorem and its corollary, each multi-dimensional surface consists of rays pointing up and out from $(g, r, G, R, m) = (1, 1, 0, 0, 1)$. Sliding figure 5.5 in toward the r origin causes the curves to retain their shape, but shrinks them in the R and MCPR dimensions.

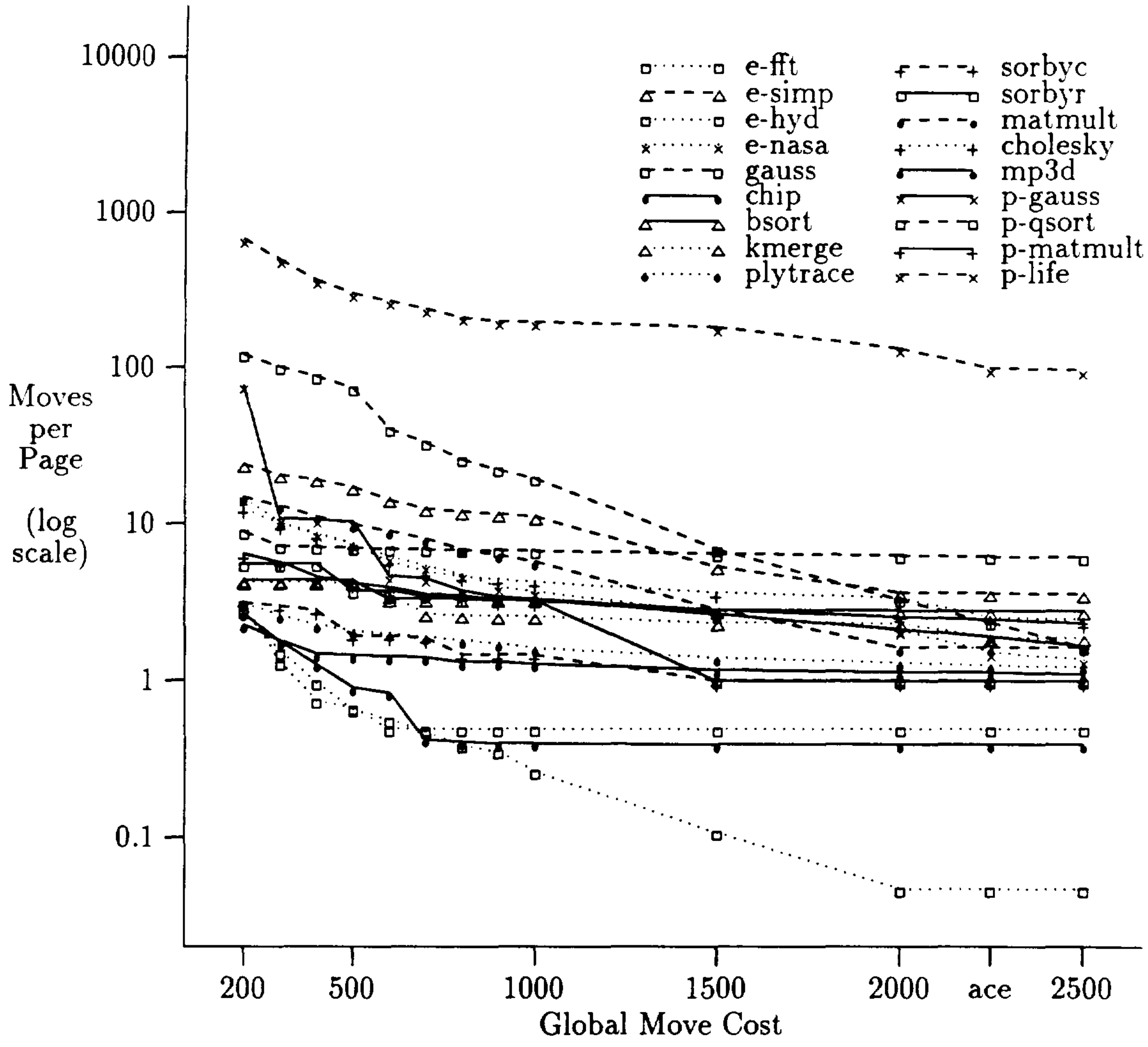

Figure 5.6: Mean page moves per 4 kilobyte page for optimal, g=2, r=5

Figure 5.6 presents, on a logarithmic scale, the mean number of page moves per page as a function of G for an ACE-like machine. Many of the applications have large jumps in the number of moves made around 1024 and 512. These are points at which referencing each word on a page, or half of the words, is sufficient to justify a page move. Some applications show large jumps at other multiples or fractions of the page size, but large changes at other values of the page move cost are rare.

When designing a NUMA policy for a given machine, one should take into account where on the move cost spectrum the architecture lies. Machines to the left of jumps benefit from more aggressive policies, machines to the right from more conservative policies. A machine that lies near a jump point will run well with policies of varying aggressiveness.

5.3.1 Case Study: Successive Over-Relaxation

To illustrate what is happening to the optimal placement as the page move speed varies, this section examines one of the successive over-relaxation (SOR) applications, `sorbyr`, in some depth. Recall from section 2.2 that `sorbyr` is a program for computing the steady-state temperatures of the interior points of a rectangular object given the temperatures of the edge points. It represents the object with a two-dimensional array, and lets each processor compute values in a contiguous band of rows. Most pages are therefore used by only one processor. The shared pages are used alternately by two processors; one processor only reads the page, while the other makes both reads and writes, for a total of four times as many references.

Almost all of `sorbyr`'s references are to memory that is used by only one processor. Thus, the MCPR values are all close to 1. However, this case study concentrates on the portion of references that are to memory that is shared. The effects of management of this memory are still clearly visible in the results presented and are typical of shared memory in other NUMA applications.

The optimal placement behavior for a shared page depends on the relative costs of page moves to local, global and remote references. This behavior is illustrated in Figure 5.7 as a function of page move cost. In this graph the cost of the optimal policy is broken down into components for page moves, remote references, global references and local references. Since most pages are used by only one processor, the major cost component is local references; in this figure, however, the local section is clipped for readability.

As page move cost decreases, remote references are traded for copies and global references, and then for more copies and local references. This can be seen in Figure 5.7 at points near $G = 1200$ and $G = 400$ respectively. It is important to note that while the cost breakdown of the optimal policy undergoes large sudden changes, the cost itself as a function of G is continuous.

The performance of the other policies is also included. The PLATINUM policy works best for small G. This is expected, since it is designed for a machine with a relatively fast page move. However, since it must be above optimal at all points and is a straight line (*i.e.*, is architecture insensitive), it must be bad for large G

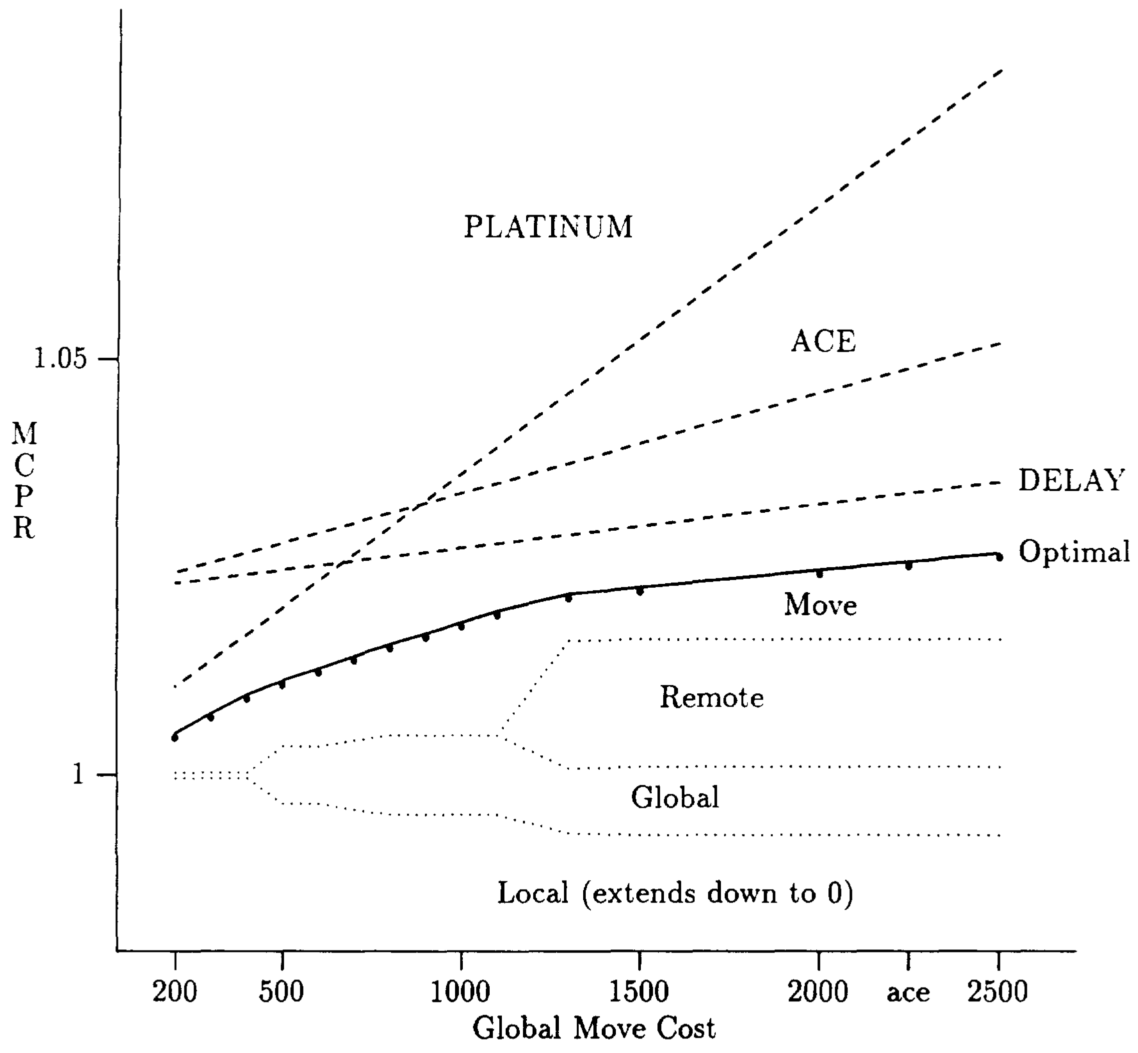

Figure 5.7: **sorbyr** MCPR vs. page move cost with optimal breakdown, g=2, r=5

in order to be good for small G. Conversely, for the ACE or Delay policies to do well for large G they must not do as well for small G.

5.4 Summary

This chapter addressed issues in the design of kernel-based NUMA management policies. The approach was to use multiprocessor memory reference traces, from a variety of applications, to drive simulations of alternative policies under a range of architectural parameters. In the area of NUMA policy design:

- The problem is important. In comparison to naive placement of shared data, optimal placement can improve overall program performance by as much as 25 to 50%. This is much higher in machines with higher remote latency, such as those studied in Chapter 6.
- Good performance on NUMA machines depends critically on appropriate program design. Trace analysis supports the intuition that NUMA policies will achieve the best performance for applications that minimize fine-grain sharing and the *false sharing* that occurs when data items accessed by disjoint sets of processors are inadvertently placed on a common page.
- Given good program design, simple kernel-based NUMA policies can provide close to optimal performance. Averaged over all applications on the ACE, the ACE policy achieved 82% of the savings achieved by the optimal algorithm over static global placement of shared data. Averaged over all applications on the Butterfly, the PLATINUM policy achieved 94% of the savings achieved by the optimal algorithm over random placement of shared data.
- Different memory architectures require different policies for high-quality NUMA management. Dynamic discovery of a good static placement works well on a machine in which page movement is expensive in comparison to the cost of remote (or global) access. As the cost of page movement decreases, it becomes increasingly profitable to move pages between program phases. The PLATINUM policy achieved only 40% of the optimal improvement on average on the ACE. The ACE policy achieved only 87% of the optimal improvement on average on the Butterfly.

In terms of architectural design:

- A policy that is (wisely) unaggressive about dynamic page placement due to high page move cost could use a mechanism such as the proposed per-page reference counter to significantly reduce the number of page-placement errors, thereby improving performance. On average on the ACE, the Delay algorithm achieved an additional 4.6% of the improvement of the optimal algorithm. On the Butterfly, it prevented two disastrous mistakes.

- Improving block transfer speed by some multiplicative factor f can lead to an improvement of more than f in memory cost, because a good policy is likely to move more pages when doing so is cheap. At the same time, there are no points at which a small improvement in block transfer speed produces a large improvement in memory cost. Fast block transfer will be most effective on machines in which remote memory references are comparatively expensive, and then only for certain applications; it is unlikely to be cost-effective on a machine with cheap remote memory.

6 Comparative Performance of Coherently Cached, NUMA, and DSM Architectures on Shared Memory Programs

Previous chapters set the stage for the work that directly addresses the thesis statement: "Properly designed software-coherent shared-memory multiprocessors can perform competitively with coherently cached machines." This chapter uses the tools and lessons of what precedes it to demonstrate the thesis.

Specifically, the aim in this chapter is to evaluate the relative merits of the three principal architectural alternatives for shared-memory parallel processing: hardware cache-coherent multiprocessors, software coherent NUMA multiprocessors, and multicomputers running distributed shared-memory systems. For the architectures that have remote access, and so require policy decisions to be made, optimal analysis is used to avoid the potential bias of using policies that are better tuned to one architecture than another.

All of the machine models have similar processors and caches, an inter-processor memory connection medium of equal bandwidth and latency, and for those machines that have it, remote memory reference hardware of the same speed. The differences lie in the way in which the components are used: do they provide hardware support for remote memory access; do they implement coherence in hardware; what size are the cache lines or pages?

The results here apply to the extent that the base technology is as described in section 6.1.2, that machines are programmed in a shared-memory style, that the machines enforce sequential consistency and are not latency-tolerant, and that latency dominates contention as a contributor to memory system cost. Machines with weak consistency reduce the number of remote operations necessary, in part by reducing false sharing, and so would benefit less from smaller block sizes; they still need to wait for events that constitute real communication events. Machines that are latency-tolerant by virtue of fast thread context switching [3] suffer less from expensive remote operations, but would still benefit from having fewer such operations by reducing false sharing. Because most remote operations are probably due to false sharing, reducing it by having smaller blocks, better compilers or better programs would benefit all of the machine models to some extent or another.

That said, most of the programs that perform well on cache-coherent machines perform only marginally worse on NUMA machines with small block sizes. DSM machines with small block sizes perform somewhat worse than NUMA machines. The block size (i.e. the size of the cache line or page) is the dominant factor in all three classes of machines; smaller blocks perform better than larger ones (within reason) in most applications, despite the fact that communication start-up latencies make moving large amounts of data far more expensive using smaller blocks. Remote references do little to improve the performance of machines with small, hardware-coherent cache lines, but they can improve the performance of distributed-memory machines dramatically, even when implemented in software.

The next section describes the machine models. Section 6.2 describes the experiments and their results. Section 6.3 examines the effect that reducing the page size in NUMA machines would have on TLB miss rates. Section 6.4 summarizes the results and presents conclusions.

A preliminary version of this work appears as [25].

6.1 Machine Models

Since the goal is to compare the performance of a suite of programs on several classes of multiprocessor architectures, it is necessary to choose machines that are representative of these classes, but which are in other ways balanced so as to avoid tainting the results with effects unrelated to the cache and memory systems used. Because such a large group of well-balanced machines does not exist (even as concrete designs), the performance of the components is extrapolated from current machines.

6.1.1 The Machine Models

There are five basic machine types. Four are obtained by considering the four pairs of answers to two questions. The first question is: "Does the machine support single-word references to remote memory?" and the second: "Does hardware implement all of the block movement decisions and operations, or does the operating system kernel need to be invoked in some cases?" Table 6.1 shows the name given to each pair of answers to these questions.

	Hardware Coherence	Software Coherence
Remote Access Hardware	CC+	NUMA
No Remote Access Hardware	CC	DSM

Table 6.1: Machine types considered

"CC" stands for (hardware) coherently cached, "NUMA" for non-uniform memory access (*i.e.*, software coherent), and "DSM" for distributed shared memory. The "+" in CC+ indicates that this model contains a feature (remote references) not normally present in cached machines.[1] The NUMA machine includes hardware to remotely reference memory in another processor's cache, but to initiate a page move operation requires a page fault; the kernel intervenes to initiate the operation to fetch the page from remote memory. CC+ is similar to NUMA, except that it has hardware support for making the remote move or replication request, and consequently does so much faster. CC is similar to CC+, except that remote references are prohibited. DSM is similar to NUMA, except that remote references are prohibited. All of the models have infinite size caches, and ignore initial cache loading effects.

While DSM does not support remote memory access in hardware, it could be implemented in software. The resulting system is called DSM+. In DSM+, a page that is to be accessed remotely is left invalid in the page table, so that every reference generates a fault. The kernel sends a message to read or write the remote location, and restarts the faulting instruction when the remote operation completes. The cost of a remote reference will be larger than in the NUMA machine, due to this kernel overhead.

Systems without remote reference capability (CC, DSM) make no placement decisions; they always migrate data on a write and replicate them on a read. The optimal placement is used for the other three models.

Regardless of the trace used (and regardless of the particular machine component speed parameters used, presuming that they are all greater than zero) some of the machine models will always perform better than others at the same block size. This performance domination relationship forms a partial order, which is shown in Figure 6.1. Machines with better performance are above machines with worse: CC+ always does at least as well as all other machines, and DSM does no better than any other. NUMA is at least as good as DSM+. To see why these relationships hold, consider that CC+ is like NUMA, but with faster remote

[1]Many cache-coherent machines allow caching to be disabled for particular address ranges, forcing processors to access data in main memory. Machines with this capability will display performance in between that of CC and CC+.

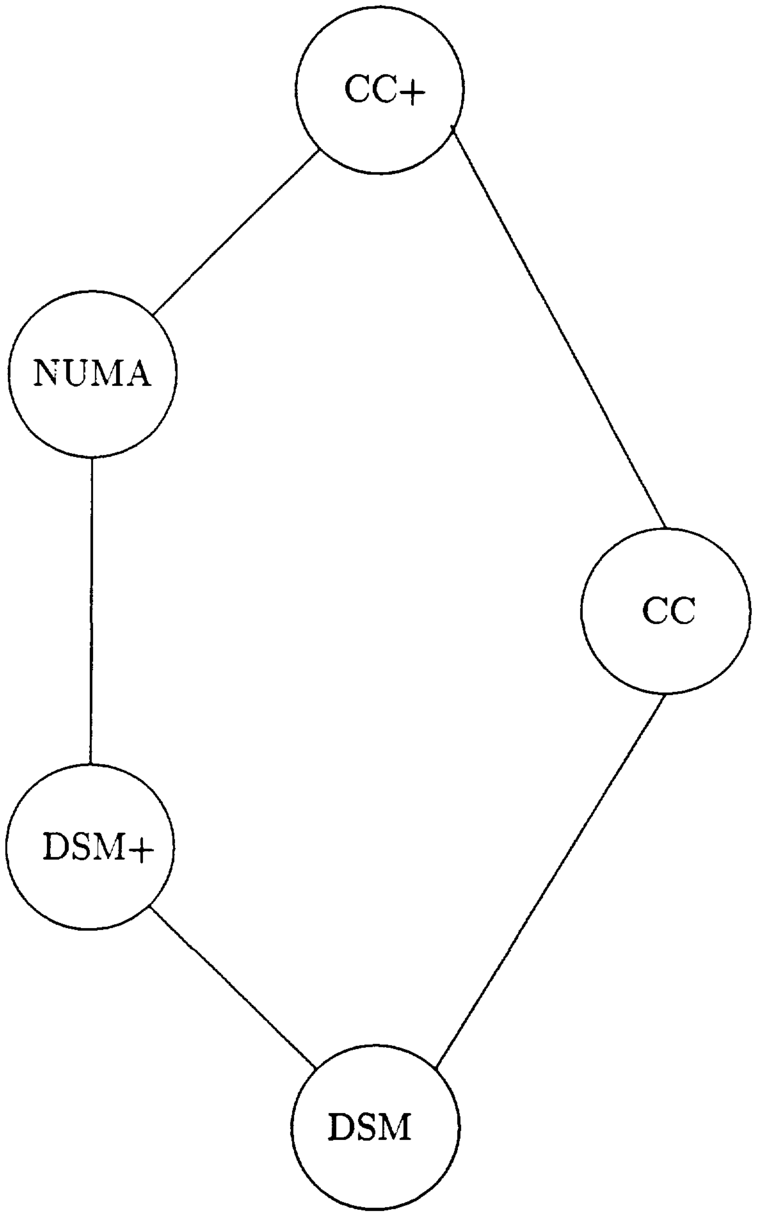

Figure 6.1: Performance domination partial order for a fixed block size

C-2

reference and block move; NUMA and DSM+ have equal block move speeds, but NUMA has faster remote reference. CC and DSM both lack remote reference, but CC has faster block move. A machine with remote reference will be at least as fast as one with equal block move, but no remote reference, because in the worst case the remote reference capability can just be ignored. Therefore, CC+ dominates CC and DSM+ dominates DSM. The final relationship, that between CC and DSM, holds because DSM has a slower block move than CC, and neither have remote reference.

The effect of block size is more complicated. While reducing the block size means that large transfers will take more operations, and so use more overhead, a small block size reduces fragmentation and false sharing [35, 41, 42]. Software managed pages are much larger than cache lines, because the time to handle a page fault is generally much larger than that to handle a cache miss. In uniprocessor systems, the tradeoff in pagesize is between internal fragmentation and increased overhead, and the trend has been to larger pages. This trend has been driven largely by the increase in system physical memory sizes, which has reduced the magnitude of the performance effect of fragmentation, while leaving the overhead unchanged. However, the added effect of false sharing reduces the desired blocksize on shared memory multiprocessors.

6.1.2 Computing Cost Numbers for the Machines

To use the cost model (see Chapter 4) the important characteristics of a machine are: how fast can it move a cache line or page from one processor to another; can it reference a single word remotely, and if so how fast; how big are its cache lines or pages? That is, what are R, r and the block size?

While only these three parameters are important to the model, deciding their values requires a more detailed understanding of what happens inside a machine.

The "local" time used as the unit in the cost model is the local cache hit time. The baseline processor is a MIPS R3000 running at 40 MHz, with one wait state for a local cache hit. Thus, a cost of 1 time unit can roughly be considered to be 50ns: one 25ns processor cycle for the load or store instruction and one waiting for the cache. The inter-processor interconnection network in all of the machines is identical in performance: it has a bandwidth of 40Mbytes/sec, and thus requires 2 time units to transmit each four bytes of data, once start-up overhead has been paid.[2]

[2]For simplicity's sake, the interconnection network is considered to be no more than four bytes wide. While larger sizes are common, it is unlikely that they would significantly affect r, or have any effect on R at all. If the interconnection were more than four bytes wide, r would be larger, because some bandwidth would be unused when transmitting a single word reference.

To model large machines, the latency of the network is 50 cost units (2.5μs) to get a message from one node to another—100 for a round trip message. The one-way latency is called λ.

The R3000 takes about 130 processor cycles to take a trap and return to the user context [8]. Because a cache cycle is two processor cycles, it costs 65 cost units to take and return from a trap. A well-coded kernel should be able to determine what action to take within another ten units, so the time charged for a trap to software to initiate a remote operation is 75 units. Any additional processing (e.g. to modify page tables) can proceed in parallel with the network transfer. On the other hand, a cache controller is likely to be able to decide what action to take on a cache miss in much less time; the model charges only 2 units for the hardware to decide what action to take on a cache miss. These two costs are named the "software overhead," o_s, and "hardware overhead," o_h, respectively.

The operation of moving a remote cache line or memory page to the local processor consists of several distinct phases: taking the initial miss or fault that begins the operation, determining the location of the memory to be fetched, requesting that the memory be sent, waiting for the memory to arrive (while concurrently updating any TLB, page table or cache directory information that may need it), and returning to the process context. The trap, decision and return cost are discussed in the previous paragraph: they are 75 for software implementation and 2 for hardware. Only one trap overhead is charged, because the hardware on the side that holds the data is assumed to be able to send it without requiring software intervention, even for the software coherent machines. All of the machine models assume that blocks have both *home* and *owner* locations, and that a distributed directory contains, on each home node, the location of the owner. For the software coherent machines to fetch a remotely located block, they have to first determine the location of the block by querying the block's home node at a cost of 2λ, and then requesting the data from the owner, at a cost of 2λ plus the time to transmit the block of block_size/2. (The transmission time is $\frac{1}{2}$ times the block size because the interconnection is able to transmit two bytes per time unit, as described above.) Together these terms determine R for the software coherent machines: $R = \text{block\_size}/2 + 4\lambda + o_s$.

In the hardware coherent machines, the cache directory at a line's home node is able to forward the request for the line's data directly to the owner node, thus eliminating one of the trips through the interconnection (and hence subtracting one λ from the cost). This results in the hardware coherent machines having $R = \text{block\_size}/2 + 3\lambda + o_h$.

The second model parameter is r, the time to fetch a single word from a (known) remote memory. Here, hardware support is provided for this operation in CC+ and NUMA. After an initial o_h hardware set-up cost, retrieving the data requires a request/reply latency time of 2λ. In DSM+, every remote reference

Machine	r	R
NUMA	$2\lambda + o_h$	$4\lambda + \text{block\_size}/2 + o_s$
CC	—	$3\lambda + \text{block\_size}/2 + o_h$
CC+	$2\lambda + o_h$	$3\lambda + \text{block\_size}/2 + o_h$
DSM	—	$4\lambda + \text{block\_size}/2 + o_s$
DSM+	$2\lambda + 2o_s$	$4\lambda + \text{block\_size}/2 + o_s$

Table 6.2: Formulas for computing model parameters

Machine	Block Size	r	R
NUMA	4K	102	2323
CC	64	—	184
CC+	64	102	184
DSM	4K	—	2323
DSM+	4K	250	2323

Table 6.3: Parameter values for the base machine models

must go through the kernel trap mechanism. After taking the trap, the kernel sends a message to the (known) holder of the memory, who traps, finds the data and responds. Thus, there are *two* software trap costs (one on each side) plus a request/reply latency. The total cost of a remote reference is $2\lambda + o_h = 102$ in CC+ and NUMA, and $2\lambda + 2o_s = 250$ in DSM+. In the machines without remote reference (CC and DSM), r is infinite.

The following experiments vary several of the parameters, and in particular explore the effect of changing the block size. The "base machine models" assume that the block size is 4 kilobytes for NUMA, DSM and DSM+, and 64 bytes for CC and CC+; these values are similar to those of machines currently being built. In particular, the page size of the ACE on which the traces were collected was 4K, and so the applications in the test suite tended to allocate their data on 4K boundaries. This explains the significant degradation in performance seen between 4K and 8K block sizes in many of the applications. Table 6.3 presents the values of the model parameters for the base systems.

6.2 Experimental Results

This section describes a series of experiments. The first is to simply look at the performance of the base machine models on the application suite. This investigation reveals that CC and CC+ almost always outperform NUMA, DSM and DSM+. To attempt to determine if this is related to having faster primitive operations or to having a smaller block size, a second experiment considers the relative performance of the machines when all five have the same (intermediate) block size. This experiment yields qualitatively different results from the base machine models: the remote reference capability of NUMA generally more than overcomes the faster hardware of CC; this means that the advantage seen in the base machines models is due to a smaller block size (and so less false sharing and internal fragmentation), rather than the faster hardware operations.

The next set of experiments explores the effect of changing the block size over a wide range of values for all the architectures, and confirms that the size of the block size effect is usually much larger than that due to the various architectural differences. The final experiments explore the effect of the (fairly optimistic) choice of o_s, the time for software to handle a page fault, and conclude that increasing o_s by an order of magnitude would leave unchanged the basic result that block size affects performance much more than architectural differences.

6.2.1 Base Machine Model Results

Figure 6.2 shows the performance results for the base machine models. The performance of each application is shown as a group of five bars, one for each architecture. The bars represent the architectures in order, showing CC+, CC, NUMA, DSM+ and DSM from top to bottom. The length of each bar shows the MCPR of the application on that architecture. For application/architecture pars that had an MCPR of greater than 35, the bar is left open at the right hand side and the numeric value is printed there. For example, DSM on `e-hyd` had an MCPR of 75.

The most important observations to be drawn from this experiment are that in most cases CC outperformed NUMA, often by a considerable amount, and that the gaps between NUMA and DSM+ and DSM+ and DSM were often also large.

Consider the second observation first: NUMA significantly outperformed DSM+, which in turn significantly outperformed DSM. That they came in this order is unsurprising, given their positions in the partial order shown in Figure 6.1. What is interesting is the magnitude of the difference. The gap between NUMA and DSM+ shows the advantage of having remote reference implemented in hardware rather than using much slower page fault mechanisms. NUMA and DSM+ performed a considerable number of remote references in the course of executing these applications; while DSM+ was able to reduce the effect of having a larger r by transforming some of the marginally valuable remote references into page moves (which cost the same in DSM+ and NUMA), DSM+ still showed large degradations in many cases. This says that at the 4K page size, remote reference is being used often and to great advantage. DSM+ is about 2.5 times slower

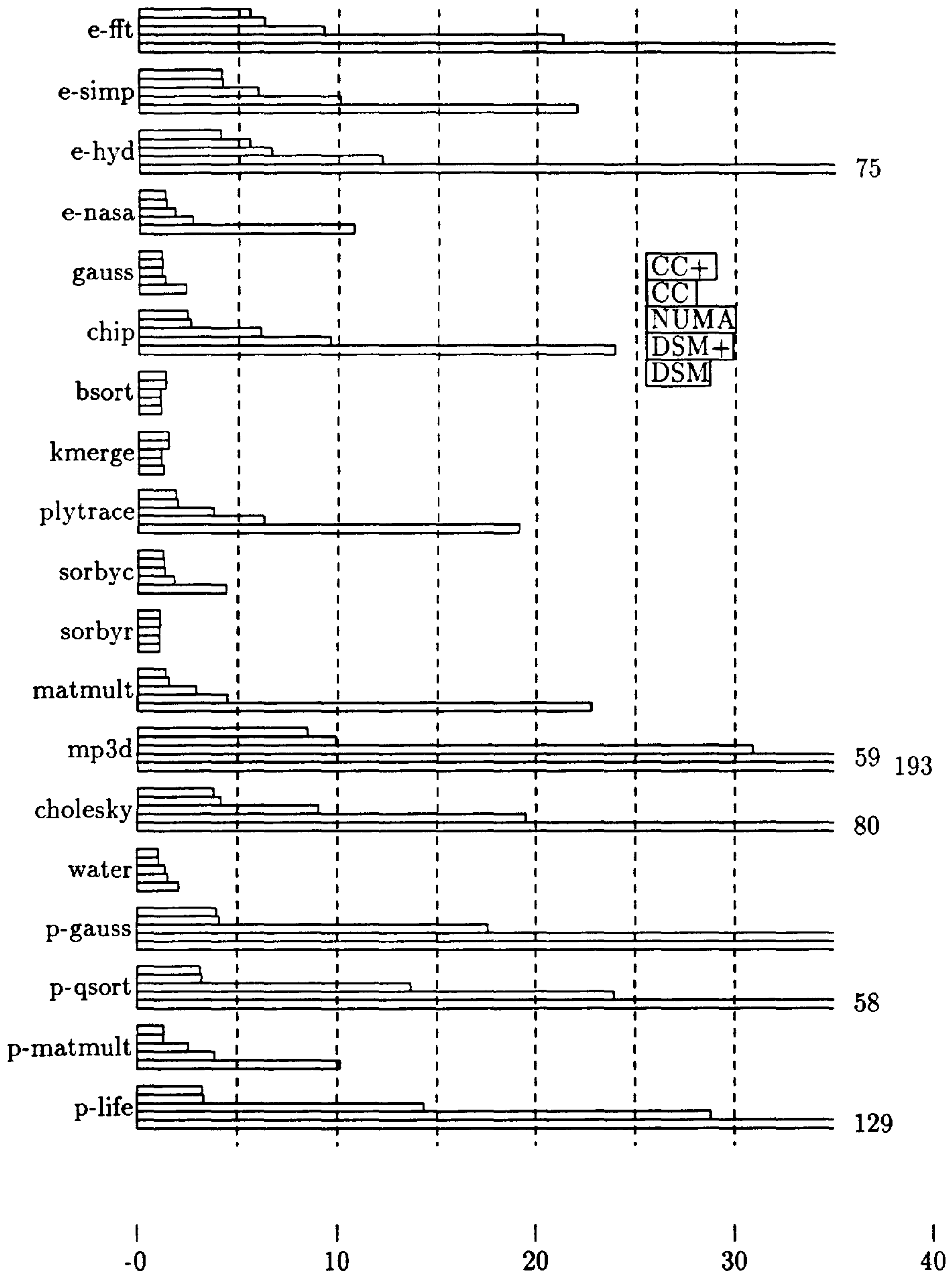

Figure 6.2: Results for base machine models

than NUMA at remote reference (102 vs 250; see Table 6.3). In several of the applications, for example **mp3d** and **p-matmult**, DSM+ MCPR was nearly twice NUMA's, which indicates that remote reference was being used for most of the interprocessor communication in NUMA.

The gap between DSM+ and DSM is also indicative of the value of remote reference. To explain this, consider a graph of MCPR versus r for a particular application. As r increases, MCPR increases less than linearly: as remote references become more expensive, they are replaced by page moves. The curve is piecewise linear, and bounded by the MCPR of DSM. It reeaches the value of DSM at the point where the optimal placement uses no remote references, and is constant for all r above that point. The fact that DSM+ does much better than DSM indicates that the particular value used for r is still sufficiently small that this curve is well away from its limiting value. For pages as large as 4K, supporting even expensive remote references pays off.

The more interesting question is that of the relative performance of the CC models with NUMA. Here, both CC+ and CC do much better than NUMA, pretty much across the board. Because of the difference in page size, the dominance relationship shown in Figure 6.1 does not hold, and in fact NUMA outperformed CC+ in **gauss**, **bsort**, **kmerge** and **sorbyr**, though only by a small amount. The question is, why do CC+ and CC do so much better overall than NUMA? They have faster operations, and a smaller page size. It is not immediately clear which difference is mostly responsible for the effect seen. While R (and r in the case of CC+) has less of an overhead component than NUMA, it seems like too small of a difference. On the other hand, one might expect that moving data in large blocks could produce significant advantages, because the overhead is amortized over a larger amount of data. Moving 4K using NUMA, or either of the DSM models with a 4K block size costs 2323 cost units. Moving it with CC or CC+ using 64 byte blocks costs 11,776. Reduction of fragmentation and false sharing would tend to benefit machines with smaller block sizes.

6.2.2 Comparing the Machine Models Using the Same Block Size in All

To shed some light on the question of why CC and CC+ performed better than NUMA, I ran a second experiment. This time, the simulations were run using a 512 byte block for all five architectures.[3] Figure 6.3 shows the result of this experiment.

In sharp contrast to the first experiment, NUMA performs nearly as well as CC+, and in most cases better than CC. This indicates that the value of faster

[3]The reason for using 512 byte blocks was that it is (logarithmically) half way between the 64 byte and 4 kilobyte sizes used in the first experiment.

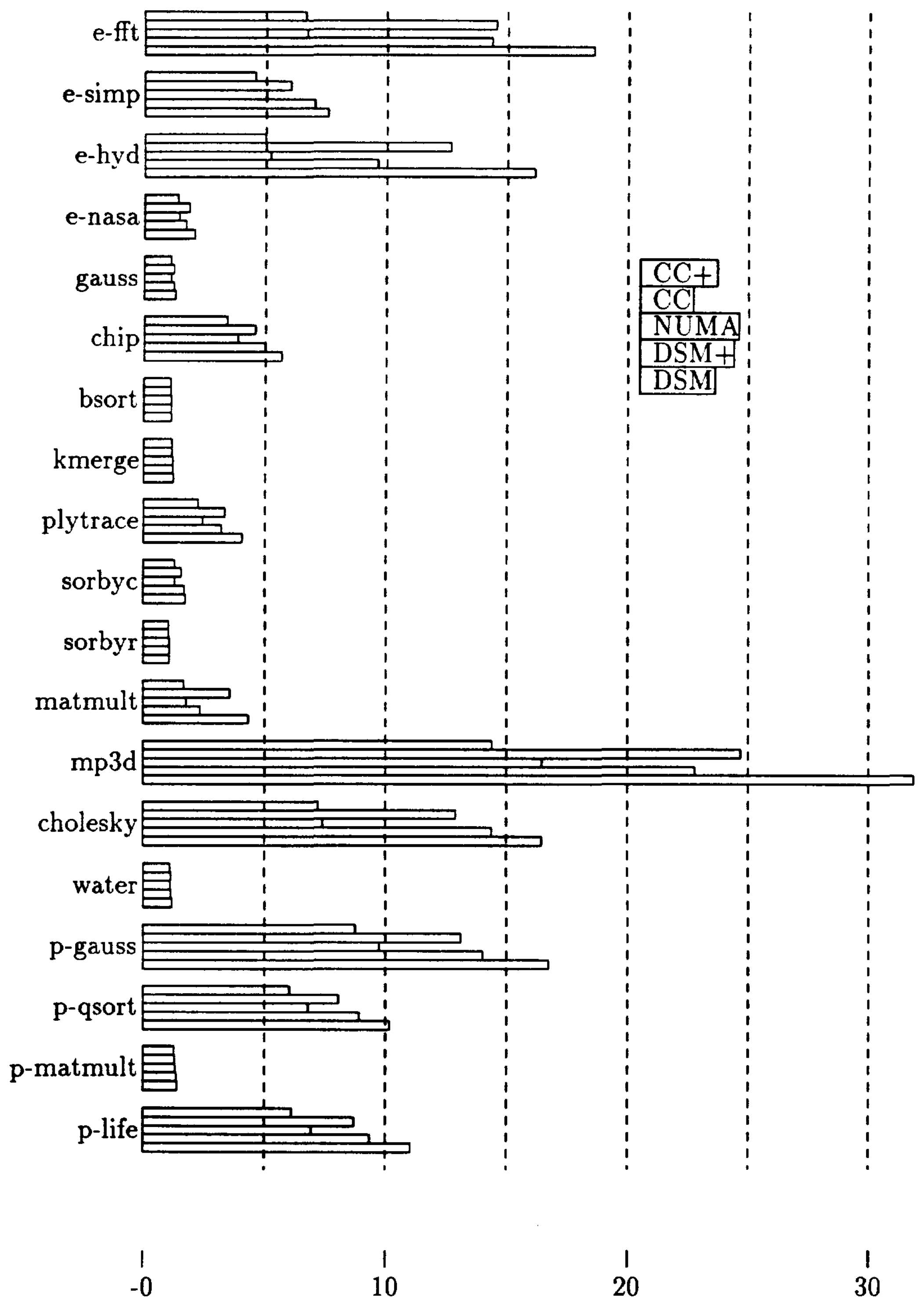

Figure 6.3: Results using 512 byte blocks for all machines

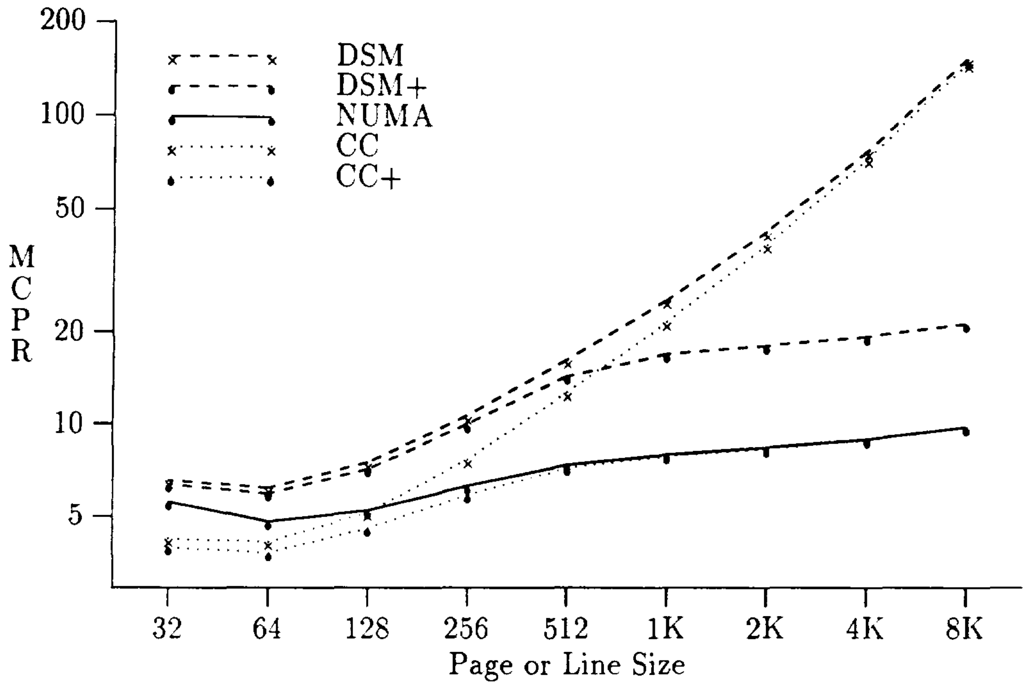

Figure 6.4: `cholesky` MCPR versus block size (log scales)

operation initiation (at least at this pagesize) is not that large, and in fact mostly smaller than the value of having remote reference. This answers the question posed by the previous experiment: the main reason that the software coherent machines performed much worse than the hardware coherent machines was not that the hardware coherent machines could initiate operations more quickly, but rather that they benefited greatly from having a smaller block size.

Chapter 7 discusses false sharing, and explains that false sharing is probably the cause of the substantial benefit of having smaller blocks. In short, having smaller blocks means that there will often be fewer coherence operations, because the system is better able to differentiate between operations that might involve the real communication of data and those that cannot possibly, because they reference different words in memory. Since fewer unnecessary (for program correctness) coherence operations take place, program execution is faster. This is true even though the cost of moving large contiguous blocks of memory is much greater.

6.2.3 Varying the Block Size

Figures 6.4, 6.5 and 6.6 show the performance of three of the applications as the block size is varied. These are log-log graphs. **`Cholesky`** and **`p-qsort`** are typical of the rest of the application set in that performance improves significantly with reduction in block size, across the board (excepting the smallest cache line size considered). The value of remote reference (as shown by the distance between

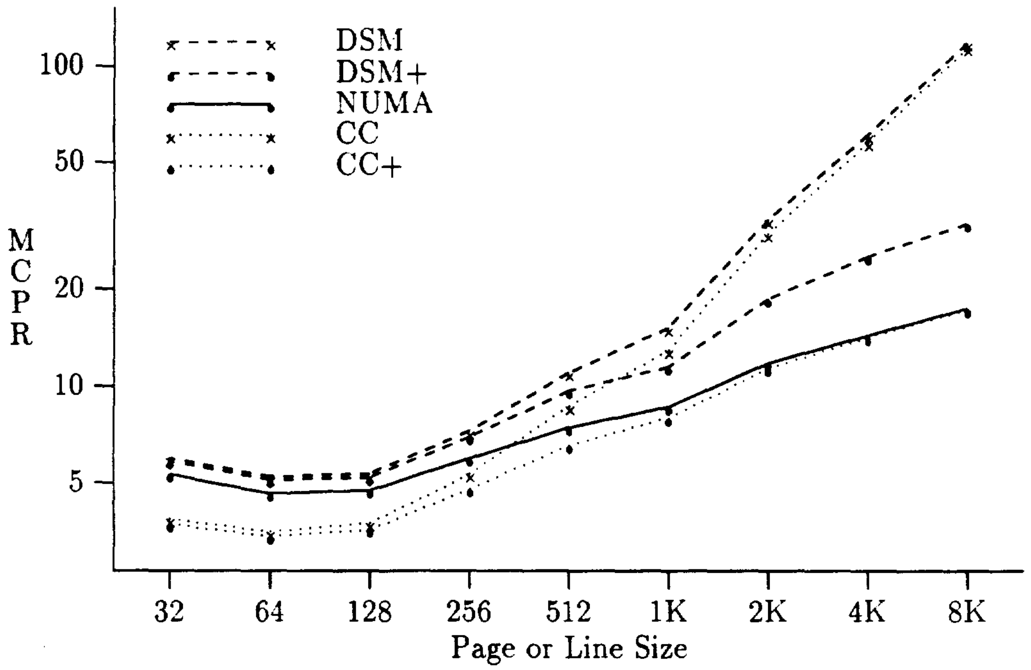

Figure 6.5: p-qsort MCPR versus block size (log scales)

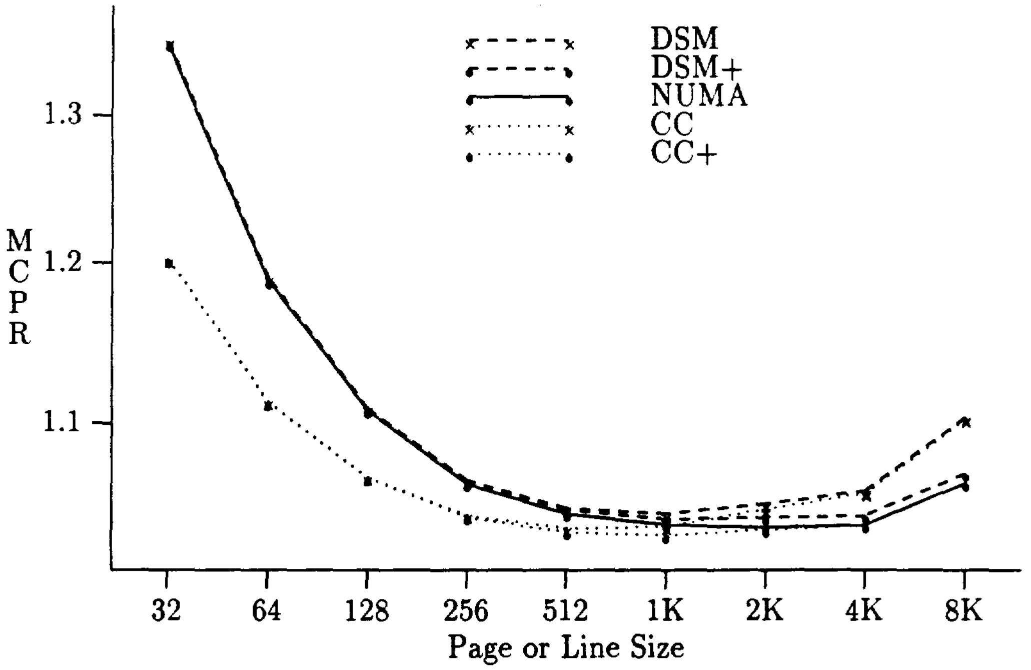

Figure 6.6: sorbyr MCPR versus block size (log scales)

DSM and DSM+ and between CC and CC+) increases with the block size. This is because r does not depend on the block size, while R does, so R/r decreases with the block size.

Sorbyr is interesting in that it is extremely well-behaved. Most of its memory is not shared, and the portion that is shared is migratory over relatively long time periods. The size of the portions that migrate is 4K bytes, and thus the curves show minima between 1K and 4K. For all machines and all block sizes, sorbyr does much better than most of the other applications. Moreover, the shapes of its curves are unique. At small block sizes performance degrades markedly, because the natural size to migrate is 4K, and the overhead of using many high latency operations rather than a single one to make the migrations dominates the application's overall performance. The relative expense of moving large amounts of data in the small-block systems led me to believe that this effect would appear in more applications, but it did not: in most applications, the advantage of reducing false sharing was more important that the added overhead introduced by moving data in small chunks.

For completeness, graphs of MCPR versus block size for the rest of the applications appear in Appendix A.

Table 6.4 shows the performance of CC, NUMA, DSM and DSM+ relative to CC+, using the lowest cost blocksize for each application/machine pair. For example, NUMA's best MCPR for sorbyr came at block size 2K and was only slightly over CC+. DSM's best was at a 1K block size and was 1% worse than CC+. This table shows that for the most part, CC and NUMA performed within 20-30% of CC+, and DSM+ and DSM were generally within 50%. Contrast this with the data presented in Figure 6.2, which shows NUMA and the DSM's commonly performing an order of magnitude or more worse than CC+ and CC.

6.2.4 The Effect of o_s

One of the parameter selections that is subject to debate is that of o_s, the software overhead. The value chosen represents an extremely small number of instructions executed by the kernel to decide whether to replicate a page (about 20, excluding those necessary in an "empty" trap). It may be that this is insufficient time; even if it is possible, it is doubtless more convenient for the kernel writer if more time is allowed. To see how sensitive the results are to o_s, the following experiments observe the result of varying it.

Across the application suite, variations in o_s resulted in two qualitatively different patterns of variation in performance. One pattern is typified by sorbyr, shown in Figure 6.7. The value of o_s in the base machine model is 75. As it increases NUMA, DSM, and DSM+ all show marked, and almost uniform, increases in MCPR. It is important to note the scale of the y axis, however. The only

Application	CC	NUMA	DSM+	DSM
e-fft	10%	8%	73%	77%
e-simp	5%	24%	50%	51%
e-hyd	27%	19%	75%	100%
e-nasa	7%	5%	19%	23%
gauss	4%	1%	9%	11%
chip	7%	29%	50%	51%
bsort	1%	0%	1%	2%
kmerge	3%	1%	2%	5%
plytrace	8%	18%	32%	39%
sorbyc	1%	2%	16%	17%
sorbyr	0%	0%	1%	1%
matmult	12%	10%	24%	39%
mp3d	17%	37%	68%	68%
cholesky	10%	25%	55%	62%
water	2%	2%	4%	6%
p-gauss	3%	33%	55%	55%
p-qsort	3%	50%	64%	68%
p-matmult	1%	12%	13%	14%
p-life	3%	35%	49%	51%

Table 6.4: Performance of other architectures relative to CC+ at their best block size, expressed as difference in MCPR

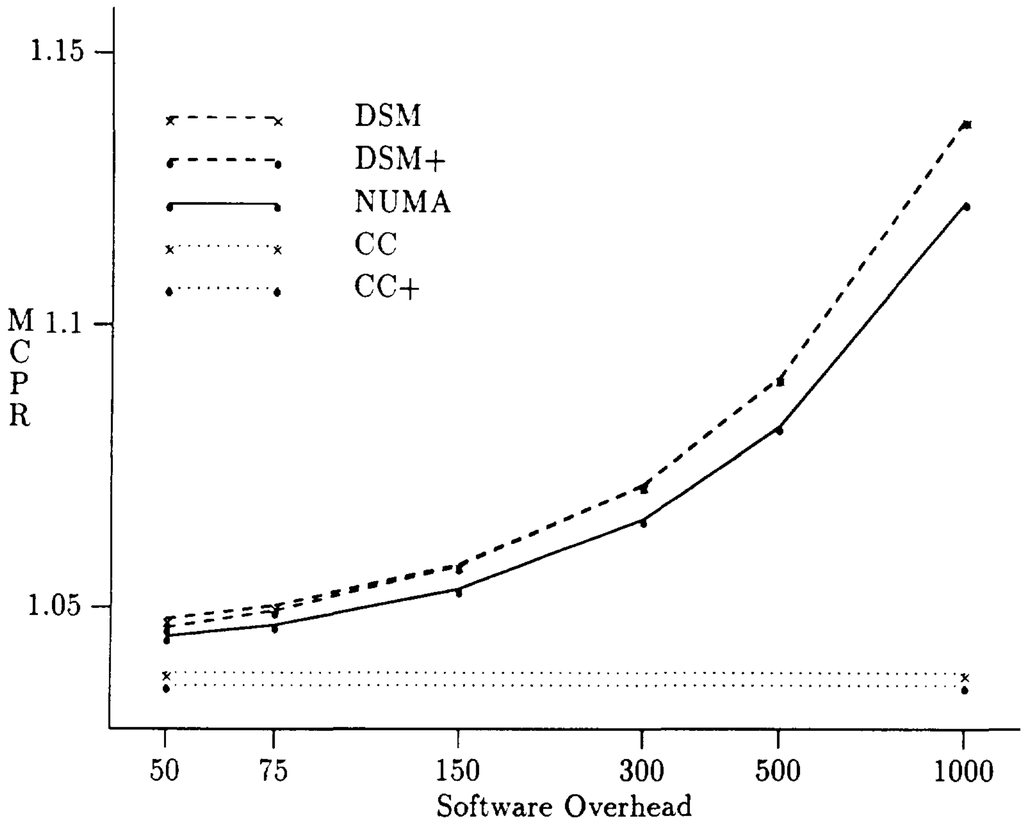

Figure 6.7: **sorbyr** performance versus o_s for 512 byte blocks (log scales)

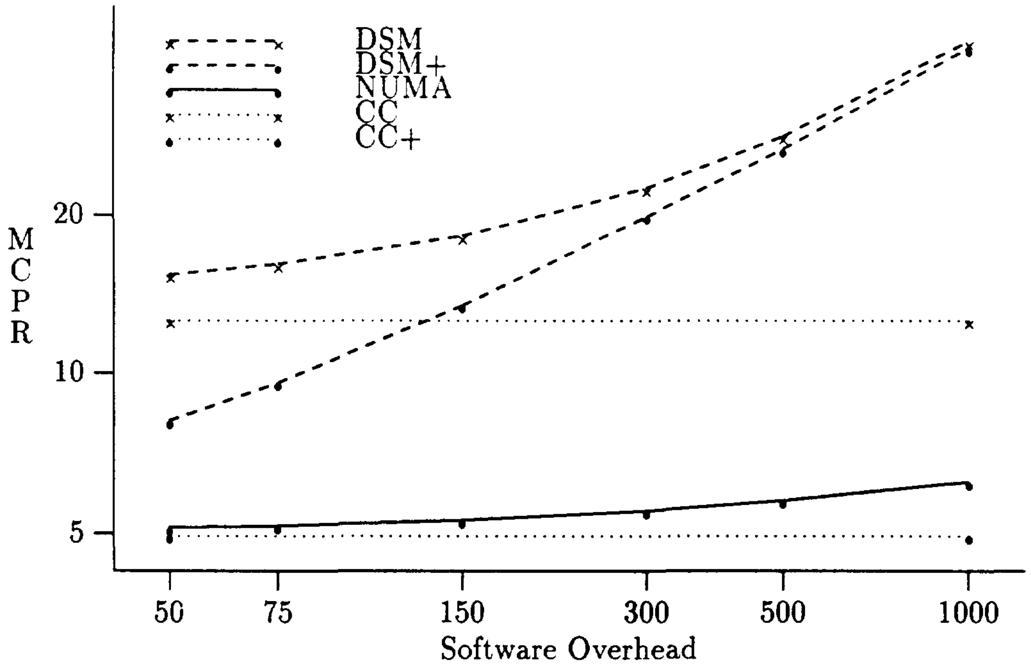

Figure 6.8: **e-hyd** MCPR versus o_s for 512 byte blocks (log scales)

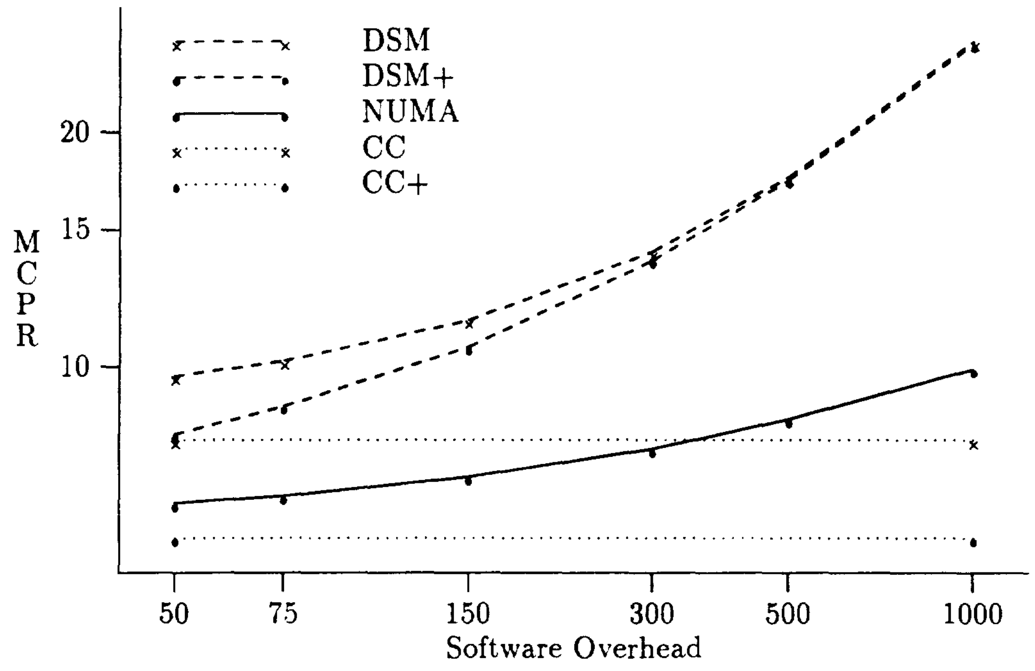

Figure 6.9: `p-qsort` MCPR versus o_s for 512 byte blocks (log scales)

applications to demonstrate this pattern of performance variation were **sorbyr**, **p-matmult**, and **bsort**, all of which have extremely good MCPR values—below 2.2—at all tested values of o_s. In other words, the shape of the graphs can be deceiving: the absolute performance degradation is small. CC and CC+ are constant with respect to o_s, because none of their operations require software intervention, and hence none of their model parameters depend on o_s. They are shown for reference only.

The more common pattern of variation of MCPR with o_s is demonstrated by **e-hyd** and `p-qsort`, shown in Figures 6.8 and 6.9, respectively. **E-hyd** is typical of the applications in this group; `p-qsort` displays the largest amount of dependence on o_s. As o_s increases the performance of NUMA slowly degrades, but not enough to be of much concern. DSM and DSM+ are more severely affected. Since software remote references in DSM+ require *two* traps, the value of remote reference in DSM+ (as evidenced by its benefit over DSM) is quickly reduced, and the two upper curves converge. With no option but to migrate blocks, and with many more migrations occurring at a cost that increases with o_s, performance rapidly degrades. Consequently, fast trap handling is crucial in DSM systems, at least when using a page size as small as 512 bytes. It is less crucial in NUMA systems, though certainly desirable, due to NUMAs remote reference capability, which is not dependent on o_s, and is able to "take up the slack" as the (o_s dependent) page move operations become more expensive.

6.3 The Effect of Reducing the Page Size on TLB Miss Rates

Using a page table to translate a virtual address into a physical address typically requires some number of references to that page table, which is in main memory. Typically, such an address translation will require two or more references. Since every memory reference by an application program requires an address translation, memory management units employ a device known as a "Translation Lookaside Buffer," or TLB, that acts as a cache of the page table. Unlike the full page table, a TLB is stored on-chip in the MMU, and is usually built from content addressable memory. When a virtual address is presented to a TLB, if the translation for that address is present in the TLB, it will be returned without paying the overhead of making references to in-memory page tables. If the given virtual address is not present in the TLB, the page tables are consulted to determine the appropriate translation, and the translation is entered into the TLB. This is known as a "TLB miss."

If the page size is reduced in a machine without virtually tagged caches, barring a change in TLB design the TLB miss rate will increase. TLBs are not part of the machine model presented in Chapter 4, and so any change in the TLB miss rate due to decreasing the page size needs to be evaluated separately before considering building a machine with smaller pages. This section presents a study in which the effect of reducing the pagesize on the TLB miss rate is calculated, and this new miss rate is used to compare the increased cost due to more TLB misses with the reduced cost due to fewer non-local operations. In most applications, the overhead of additional TLB misses is small relative to the benefit of reducing the page size.

6.3.1 Virtually Tagged Caches

Some machines with virtually tagged caches do not need to have TLBs, because the cache directory provides a substitute. That is, if the virtual address is used as the index into the cache, a virtual address translation only needs to happen when the cache misses; since cache misses are rare, and are usually more expensive than TLB misses, such a design will greatly reduce the cost of an increased TLB miss rate. However, there are several problems with virtually tagged caches. The main one is that operating systems that allow different virtual addresses to map to the same physical address need to deal with aliasing in the cache. This problem is discussed by Wheeler and Bershad [101], who find that with careful management of the cache by the operating system, the extra overhead due to the additional operations needed to eliminate aliasing problems can be small.

Other machines, such as the HP 9000 series [68], have virtually indexed, physically tagged caches, in which an address translation occurs by presenting the address to both the cache and TLB at the same time. The cache hashes the virtual address to find a line or lines that may contain the data. To determine if the line contains the correct data, its physical tag is compared with the result of the TLB lookup. This scheme allows cache indexing and TLB lookup to happen in parallel rather than in serial as in a physically indexed cache, but does not remove the need for a TLB (and so eliminate the cost of a TLB miss).

6.3.2 TLB design

TLBs are indexed in much the same way as memory caches. They are divided into k sets of size a. An address to be looked up is categorized into one of the k categories in some manner, typically by using the low order bits of the page number (and so k is normally a power of two). Once the category is determined, the a entries in the appropriate set are queried to see if they contain the translation for the virtual address. The sets are usually implemented in associative memory, so the lookup time is small. The size of the sets, a, is called the *associativity* of the cache. TLBs in which $k = 1$ are called "fully associative," because all entries coexist in a single associative store. If a TLB were built with $a = 1$, it would be called "direct mapped." TLBs that are neither direct mapped nor fully associative are called "set associative." This same terminology is used to describe associativity in caches.

This study simulates a multiprocessor in which each processor has its own TLB; references made by one processor have no effect on the TLB of another. For TLBs that are not direct mapped, when a TLB miss requires that an entry be deleted to make space for the new translation, the least recently used entry in the appropriate set is chosen for removal. For TLBs that are not fully associative, the least significant $\log_2 k$ bits of the page number are used to determine into which set a page falls. All of the TLBs studied here have values of k that are powers of two. Figure 6.10 and the similar figures in Appendix A display the TLB miss rate as a function of the page size. The miss rate itself is simply the number of TLB misses divided by the total number of references in the trace.

As noted in Table 2.1, many of the applications make references to "private" memory: memory that is *a priori* local to a single processor. For the most part, previous work has ignored these references. However, they affect the TLB miss rate in exactly the same was as do references to potentially shared memory. Therefore, the private references have been included in the input to the TLB miss rate simulator. Instruction fetches were not included.

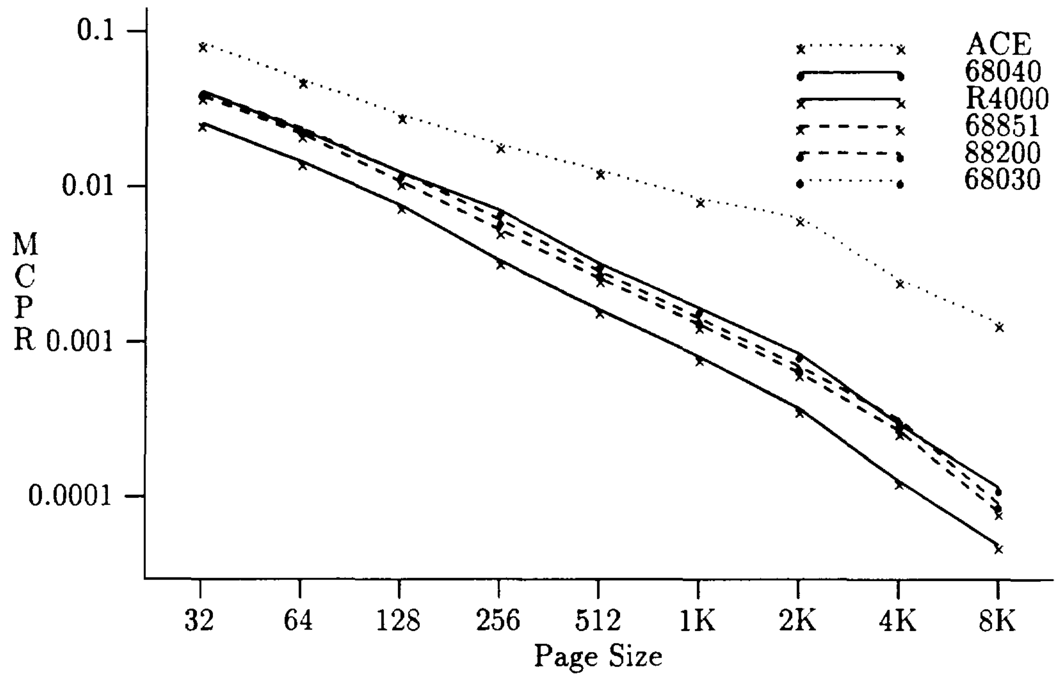

Figure 6.10: `plytrace` TLB miss rate vs. page size (log scales)

6.3.3 Results

The results presented here consider the TLB miss rate for 6 different machines. These machines are the ACE, which has 16 sets with two entries per set, the Motorola 68040 which has 16 sets of 4, the Motorola 68851 MMU, which is fully associative and has 64 entries, the Motorola 88200, which is fully associative with 56 entries, the Motorola 68030, which is fully associative and has 22 entries, and the MIPS R4000. The R4000 has 48 entries and is fully associative, but is unusual in that each TLB entry describes two adjacent virtual address translations. These translations may be to two non-adjacent physical pages, have different protections, and have separate valid bits. In this way, the associativity of the TLB (and so the expense of building the associative memory for the lookup hardware) is kept down, but it is able to cache more translations. In my model, this machine is treated as if it has twice the pagesize otherwise shown. This completely captures its miss rate, although the cost of a miss can be greater, because it requires two translations from the page table, rather than the one required by the other machines.

Figure 6.10 shows the TLB miss rate (in misses per memory reference) for `plytrace` as a function of the page size. `Plytrace` was selected because it is typical of the TLB miss behavior of the application suite. It shows significant increases in the TLB miss rate as the page size is decreased; at a page size of 128 bytes, all but the R4000 have miss rates of over 1%. For the ACE model, this is an increase of roughly an order of magnitude over the miss rate for a 4K page size, and slightly less than two orders of magnitude for the R4000. The time cost due to this increase depends on the cost to fill a TLB miss; on the ACE it is somewhere between two and three local memory reference times. Conversely, on the R4000, which has software TLB miss handling, it can be considerably more, depending on the implementation of the software filling the miss.

To fully understand the effect of the change in TLB miss rate, one needs to consider its effect on execution time, rather than simply looking at the miss rate. To compute the MCPR effect of an increased TLB miss rate requires an understanding of the cost of a TLB miss, which in turn requires an understanding of the address translation hardware of the machines involved.

The ACE has an inverted page table. The time to translate a virtual address depends on the number of memory references that have to be made into the table, which in turn depends on the length of the particular hash chain on which the virtual address being translated lies. At minimum, two references need to be made to get to the first entry on the chain. Typically, these hash chains are very short, and TLB misses only need to access one or two entries on the hash chain. Assume that the average chain is of length 2, so the average number of memory references needed for a TLB miss is 2.5. Furthermore, assume that the TLB miss rate for the 4K page case is 0. Then, the MCPR cost due to added TLB misses will be 3% (the TLB miss rate for the ACE with 128 byte pages) times 2.5 (local) references per miss times one cost unit per local reference, for a total contribution of .075. The MCPR for **plytrace** on the NUMA model at 4K pages is 3.8, while at 128 bytes it is 2.1, and so the additional cost of the TLB misses is overwhelmed by the improvement in performance due to the reduction in remote operations.

Because the R4000 uses software to fill a TLB miss, the cost of such a miss is much greater. Uhlig *et al.* [97] report that the cost to fill a user TLB miss in the Mach operating system on an R4000 is 20 machine cycles; assuming local cache hits take two machine cycles, and that the TLB miss rate for 4K pages is 0, the change in MCPR is the 128-byte page TLB miss rate of .8% times 10 (the cost of a TLB miss expressed in cache hit times), or .08 which is still insignificant compared to the improvement due to reduction in non-local references because of the 128 byte page size.

Table 6.5 shows the MCPR cost of each of the applications for the NUMA model, using 4K pages, and using 128-byte pages 1) without a correction for TLB misses 2) with a correction for the cost of additional TLB misses using a TLB model like that of the ACE and 3) using one like that of the R4000. While all of the applications suffer because of the increased TLB miss rate, only **matmult** on the R4000 and **gauss** on the ACE are severely degraded. Overall, the slowdown due to TLB effects is much smaller than the speedup because of smaller pages.

However, for applications that run on a single processor, the increased TLB miss rate can be significant. In effect, all single-processor applications have an

Application	4K NUMA MCPR	128-byte NUMA MCPR		
		Uncorrected	ACE	R4000
e-simp	5.98	5.02	5.17	5.25
e-hyd	6.64	4.84	4.89	4.87
gauss	1.14	1.13	1.41	1.16
chip	6.11	3.13	3.29	3.16
bsort	1.08	1.33	1.42	1.50
kmerge	1.12	1.46	1.53	1.56
plytrace	3.76	2.13	2.20	2.21
sorbyr	1.04	1.18	1.62	1.22
matmult	2.90	1.52	2.79	6.09
mp3d	30.9	11.7	11.8	11.8
water	1.40	1.08	1.10	1.09
p-gauss	17.6	6.18	6.26	6.21
p-qsort	14.4	4.72	4.78	4.75
p-life	14.3	4.98	5.02	5.04

Table 6.5: MCPR uncorrected and corrected for additional TLB misses

MCPR of 1, but would still suffer from the same degradation in MCPR due to an increased TLB miss rate. That is to say, a "single processor" version of `plytrace` on the R4000 would see an 8% performance degradation because of small page sizes. Furthermore, for some applications, notably `bsort`, not only does reducing the page size increase the TLB miss rate, but it also hurts the MCPR independent of TLB effects; the increased TLB miss rate simply worsens the degradation in performance already present as the pagesize is reduced in `bsort`.

These results indicate that while simply reducing the pagesize will increase the TLB miss rate, the cost of these misses for parallel programs will probably be insignificant due to the gain in performance reported earlier in this chapter. On the other hand, single processor applications will suffer from more TLB misses without the compensating benefit of better multiprocessor performance. To make single-processor programs and systems perform better one could simply enlarge the TLB, use a virtually tagged cache, have a (per-process) variable page size, or extend the R4000's idea of having multiple pages represented by a single TLB entry to a much larger number of pages per entry (say, 16). Adopting any of the solutions to the uniprocessor problem would also result in the TLB miss rate having little impact on the performance of NUMA and DSM systems as opposed to cache-coherent systems.

The TLB miss rate graphs for the other applications are in Appendix A.

6.4 Conclusions

The results of this chapter must be considered in the context of assumptions about the hardware technology, application suite, and tracing methodology. Major technology shifts could lead to different results—a large decrease in network latency, for example, would benefit the cache-coherent machines more than the NUMA or DSM systems, because their smaller block sizes cause them to incur this latency more frequently. Different applications will produce different results as well. The applications were all designed for NUMA machines or bus-based cache-coherent multiprocessors. They have been modified to represent the use of smart synchronization algorithms, but even so they do not in general display the very coarse-grain sharing most suitable to distributed-memory multicomputers. They are typical of the programs designers would *like* to be able to run on DSM systems. The cost model represents memory access latency and placement overhead only; it does not capture contention. This assumption appears to be fair in well-designed machines and applications. It may penalize the NUMA and DSM models to some degree, since contention would be more likely to occur in the cache-coherent machines. As in most trace-based studies, caches are assumed to be of infinite size, and cold-start effects are ignored. Since the machines that need to make policy decisions (the ones with remote reference) are assumed to always make the best possible choices, their actual performance will be somewhat less than that reported here; on the other hand, the machines without remote reference make no policy choices, and so their numbers show no similar bias. Finally, the fact that changes in architectural parameters should result in different traces is ignored; experiments presented in Section 4.3 indicate that these results are relatively insensitive to the kinds of trace changes that would to be induced.

With these caveats in mind, the principal conclusions are:

- Block size is the dominant factor in shared-memory program performance. Slopes at the low ends of the graphs suggest that even machines that incur large per-block data movement overheads (i.e. the NUMA and DSM systems) will still benefit from reduction of the block size. Block sizes in the 64 to 128 byte range seem to be sufficiently small that further reductions have little positive effect on performance. For particularly well-behaved applications (e.g. **sorbyr**), reductions below the 128-byte level can markedly *reduce* performance.

- Machines with coherent caching hardware tend to do somewhat better than those without. However, NUMA-style machines, particularly when run with small block sizes, perform comparably, and in some cases outperform the traditional cached machines without remote reference. NUMA machines remain a viable architectural alternative, particularly if they can be built significantly more cheaply than cache-coherent machines.

- The value of remote reference depends on the size of the block being used. For the block sizes typically employed in paged machines, it can yield significant performance improvements. Well coded remote reference software is a promising option for DSM machines, and merits experimental implementation. Conversely, remote reference is unlikely to benefit cache-coherent machines enough to warrant the expense of building it, at least when the line size is small.

- Distributed shared memory systems such as DSM and DSM+ do not appear competitive as a base for generic shared-memory programs. Their performance was acceptable only for the most well-behaved applications, with very coarse-grain sharing. At the same time, there is little hope that programmers will be able to ignore locality issues on *any* type of large shared-memory multiprocessor. Shared memory is primarily attractive as a form of global name space. It relieves programmers of the need to employ special syntax for operations on shared objects; it does not relieve them of the need to keep track of those objects' locations.

These conclusions suggest several avenues for future work in the field. Dubnicki and LeBlanc [41] have proposed that cache-coherent machines vary their block size adaptively, in response to access patterns; the results presented here suggest this is a promising idea. Hybrid architectures such as Paradigm [34], which employ bus-based hardware coherence in local clusters and software coherence among clusters, also appear to be promising; hardware coherence is easy (and cheap) for bus-based machines, and the performance of the NUMA model is acceptable for many applications. Section 6.1.1 notes that reductions in block size can worsen performance by increasing the cost of moving large amounts of data, and can improve performance both by decreasing false sharing and by increasing the probability that a block will be used enough to merit replication before it is invalidated via true sharing.

The burden of locality management currently rests with programmer. Its importance is obvious in the wide performance differences among the applications, but the task can be onerous, particularly to novice programmers. There is a strong need to reduce this burden. Fairly simple diagnostic tools could identify data structures for which coherency imposes a substantial cost at run time, thereby facilitating manual program tuning. More ambitiously, compilers or run-time packages might use application-specific knowledge to partition and place data appropriately, thereby increasing locality.

7 False Sharing and its Effect on Shared Memory Performance

Consider a typical sequentially consistent shared-memory multiprocessor. It consists of a number of processors with some form of memory or cache at each processor. This memory is grouped into blocks, and these groupings are used for purposes of maintaining coherence; a reference to any word of a particular block is treated identically to a reference to any other word in that block. Therefore, it is possible that two (or more) different processors would each use distinct memory that is located within a single block, and that this use could result in coherence operations. Since by premise each processor is using distinct memory, it is not necessary for the correctness of the program that each processor have available the other processors' up-to-date data; that is, some of the coherence operations are unnecessary. However, because of the grouping into blocks by the hardware, traditional systems will assure that this data is present at any processor that is referencing a block. Behavior wherein processors use distinct data within a block is called "false sharing." The term is also used to denote the unnecessary coherence operations performed because of false sharing; which meaning is intended should be obvious from context.

Given the concept of false sharing, several questions immediately come to mind: Can it be more precisely defined? If so, can it be accurately measured? What overall effect does it have on performance? Can it be eliminated or reduced in impact, and if so, how?

This chapter explores these questions, though it does not come to definite answers. Section 7.1 lists criteria for an acceptable definition for false sharing, and then considers several candidate definitions in term of those criteria. Each definition is found to fail on one or more of the points. Section 7.2 estimates the extent of false sharing for several application by looking at their breakdown into overhead and data transfer cost components.

Practical methods of reducing false sharing are beyond the scope of this work. The insights offered into the magnitude of the problem, however, indicate that if the problem could be solved, it would result in large improvements in program performance on shared-memory multiprocessors that have large block sizes.

One promising approach to (effectively) reduce false sharing is to change the memory model exported to the user. Instead of presenting a sequentially consistent memory model (one in which every write is immediately visible at all processors), alternate approaches could be used. For example, Stanford's DASH Multiprocessor system [52, 69] and Rice's Munin distributed shared-memory system [19, 30] use *release consistency*. In its essence, release consistency requires that newly written data be remotely visible only when the writer releases its lock on the data. In that way, false sharing generates fewer coherence operations. If two processors are operating on different objects on the same page at the same time, the number of synchronization operations necessary is no larger than the number of lock releases; these releases may not cause the processor to stall even if they cause a coherence operation, because the coherence operations take place in the secondary cache. If the processor does not use the cache line again until the coherence operation completes, the time cost of the coherence operation will be zero. Correctly constructed programs (those with no unlocked concurrent accesses to data objects) will be unaffected by the change in semantics due to weak consistency, but could perform much better.

7.1 Definitions of False Sharing

Ideally, a definition of false sharing would have the following properties:

- It would **adequately capture** the intuitive notion of false sharing.
- It would be **mathematically precise**.
- It would be **practically applicable**.

To adequately capture the intuitive notion of false sharing, the definition should result in a value that gets bigger as more unrelated things are co-located within blocks; that never grows as blocks are subdivided; and that is zero when the block size is one word. The latter two requirements taken together imply that the amount of false sharing can never be negative.

The cost and execution models as presented in Chapter 4 are all mathematically precise: given a trace and machine model, there is no ambiguity as to the cost of executing the trace on the machine. Furthermore, the framework is sufficiently rigorous to allow the proof of mathematical theorems in its context. An adequate definition of false sharing should be equally precise; if it relies on hueristic or inexact techniques, it can at best provide bounds to the amount of false sharing, and at worst inexact approximations that may lie an undetermined distance in either direction from the "true" value.

A definition that both captures the intuitive notion and is mathematically precise is in some sense sufficient: it would adequately capture the cost of false sharing in a mathematical way. However, if it is not practically applicable, because for instance applying the definition requires the solution to an NP-hard optimization problem, it is of little use.

The following sections explore some potential formal definitions for false sharing. All of them fail at least one of the above criteria. However, they are still interesting as mental exercises, and some of them are at least able to provide passable approximations to the amount of false sharing present. Section 7.1.1 describes the one-word block definition, which occurs to many people when they are first presented with the idea of false sharing, but which on closer examination badly fails to capture the intuitive notion of false sharing. Section 7.1.2 describes the interval definition, which uses future knowledge and an optimizing algorithm to measure false sharing; it fails the practical applicability test, since there is no known computationally tractable solution to the optimization problem. The next section addresses the primary weakness of the interval definition by allowing hueristic selection of intervals; it fails the preciseness criterion. Section 7.1.4 considers *full-duration* false sharing, and finds that it is too restrictive a definition. Section 7.1.5 reviews a method used by Eggers and Jeremiassen wherein a program is tuned by hand and timed to determine the extent of false sharing; it is not mathematically precise. Section 7.1.6 describes the cost-component method, which uses a breakdown of the execution cost into its constituent parts; it fails to capture the intuitive notion of false sharing.

7.1.1 The One-Word Block Definition

When the concept of false sharing is explained to many people and they are asked to quantitatively define it, after a little thought they come up with the idea that the false sharing in a trace at blocksize b is the difference in optimal cost between the trace when run with blocksize b and that when run with a blocksize of one word. Indeed, when a trace is run with a single-word blocksize there is manifestly no false sharing. Furthermore, the optimal placement for a program with a single-word blocksize will have the property that as few words are transferred between processors as is possible while maintaining coherence. However, it does not result in the minimum *number* of transfers; most programs have at least some locality of reference, and so would benefit from some grouping of transfers. The execution cost of a program as defined in Chapter 4 is the sum of the per-byte cost of transferring data through the interconnection and the per-message cost of initiating these transfers (since the one unit per local reference cost is insensitive to placement, it can be ignored). These are represented in

Chapter 6 as the "pagesize/2" and "$n\lambda + o_h$" components of R and r. Reducing the blocksize to one word minimizes the data transferred, but increases the number of operations and thus the overhead incurred. So, the naïve definition of false sharing could easily result in a negative amount of false sharing if the additional overhead generated outweighs the eliminated false-sharing induced coherence operations.

This effect can be seen very graphically in the results presented in Chapter 6. For example, Figure 6.6 shows that **sorbyr** increases in cost as the block size is lowered toward 1 word. If all that happened as the blocksize was reduced was that false sharing was also reduced, then the cost curves would all get smaller with the block size. However, exactly the opposite happens: the cost gets larger with a smaller blocksize. In this particular application, the sharing is essentially migratory in nature: four kilobyte chunks are passed between processors, and if this sharing happens by moving small pieces one at a time, performance suffers.

Increased cost through breaking up blocks that aren't falsely shared is present in most applications as block size is reduced, though it is usually not as pronounced as in **sorbyr**; most applications have a sufficient amount of false sharing that they benefit from smaller block sizes, even though big-chunk migratory data is handled poorly. This explains the general trend toward better performance at smaller block sizes seen in Chapter 6.

Because the one-word block definition of false sharing is unable to separate improvements in performance due to reductions in false sharing from degradations in performance due to increases in the number of operations needed, it fails the test for a proper definition, because it does not adequately capture the intuitive notion of false sharing.

7.1.2 The Interval Definition

Imagine that the system is granted perfect knowledge of the sharing behavior of the program (say, because the application writer inserted perfect directives into the code). Using this information, it would be possible to change the coherence constraint: instead of requiring that only one copy exist at the time of a write, only require that any time a read takes place, the reading processor sees the "freshest" data. If two processors are allowed to have disparate copies of a (logically) single block, it must be possible to re-merge these copies at some future time. Call the cost of this new merge operation M.

Define the effect of false sharing to be the difference in performance between the trace running on the original machine, and the minimal cost achievable using the extended execution model with the new merge facility. This definition is compelling in the sense that it describes a system that one could imagine implementing (given that the application writer or language tools provided good enough directives), and does not suffer the same sort of problem as the "one word

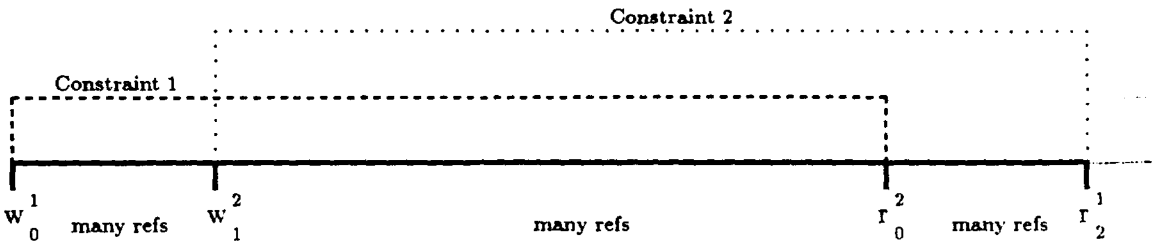

Figure 7.1: False sharing intervals

block" proposed definition. It does not require any sort of hueristic guessing, programmer induced inaccuracy or fuzziness in the size of the measured effect as do the other definitions presented later in this chapter.

However, this definition fails the third criterion for a definition of false sharing: it is not (known to be) computationally tractable. Consider a string of references to a single page made by two different processors, as illustrated in Figure 7.1. Here, a notation like r_a^p means processor p read address a. A false sharing interval, then, is any interval that contains no pair of references w_a^p and r_a^q such that $p \neq q$, the write precedes the read, and the block is written and referenced by more than one processor during the interval. That is, a false sharing interval is one in which there the block is used by more than one processor, but in which no data communication takes place. A "maximal" false sharing interval is one that ceases to be a false sharing interval if it is extended by one reference on either end, either because true data communication would be required, or because the end in question is at the beginning or end of the whole trace.

The situation shown in Figure 7.1 is that there are two potential "maximal" intervals: from just after the first write to just before the final read, or from some time in the past (determined by other constraints not shown) up to just before the first read. Neither of these intervals can be extended without violating some constraint, but yet they have a non-empty intersection and are not equal. The total number of possible "maximal" interval sets can be exponentially large in the number of references, and there is no known computationally efficient method of determining which interval set results in the lowest possible overall execution cost.

7.1.3 Heuristic Interval Selection

Given that there is no known way of optimally choosing false sharing intervals, it could still be possible to at least make a good guess as to which intervals to use. At the very least, even an arbitrarily selected interval set has the property that it provides a lower bound on the amount of false sharing present in an application. If the heuristic used is good, then the computed bound could be close to the real limit, and so would be good enough for the purposes of showing that false sharing can be a large problem.

It is possible to have a (maximal) interval that has negative value: the cost of the coherence operations eliminated by the interval are smaller than the cost of merging the block at the end of the interval. Furthermore, the cost benefit of an interval depends not only on the references made during the interval, but also on the desired starting and ending locations of the single copy of the page at the beginning and end of the interval. These locations in turn depend on what other intervals are selected. So, even the simple hueristic of choosing some set of maximal intervals, simulating the trace and rejecting those intervals that have negative value can still result in the selection of intervals that increase cost rather than reducing it, because they add cost only after other intervals have been chosen.

The hueristic interval method selection definition of false sharing fails the mathematical precision criterion; nevertheless, it is an approximation of the interval method that always errs in the direction of underestimating false sharing, and so is somewhat interesting in that provides a lower bound for false sharing.

Munin's [30] software implementation of release consistency takes a practical approach to hueristic interval selection. Its designers observe that if locking protocols are observed, any references made by a processor to an object for which it holds a lock are inside a false sharing interval (assuming that the processor does not make any references to other objects on the same page while holding the lock). This interval is not necessarily a maximal interval, but practically it will often be sufficiently large to result in a significant reduction of false sharing. After a lock release, if another processor acquires a lock for an object on the page, a merge operation happens, similar to that postulated in the interval definition (section 7.1.2). The value for M would be the time to perform such a merge.

7.1.4 Full Duration False Sharing

If an additional restriction is placed on the intervals used for defining false sharing, namely that they extend from the beginning of the trace to the end of the trace, then two helpful things happen. First, the interval selection problem goes away, because there can be only one maximal "full duration" interval. Second, the implementation question of how to deal with a program that has full-duration false sharing is much easier. All an implementor has to do, given that the full duration false sharing is identified ahead of time, is to turn coherence off for the falsely shared regions of memory. Since, by the definition of a false sharing interval and the fact that the whole of time is such an interval, no processor may read data written by another within a full-duration falsely shared block; sequential consistency will be maintained. There is no need to extend the cost model by the addition of a merge cost, M.

Unfortunately, such full duration falsely shared pages are extremely rare, and the result of exploiting this effect is sufficiently small as to be uninteresting in the applications studied here, and probably in most applications in practice. This definition fails the criterion of adequately capturing false sharing, because it is too restrictive.

Along the same lines as full duration false sharing is the identification of words that are either only-read or are used by only one processor, but are located on a page that is written by other processors. If these words could be separated at compile time, then every access to them could be local.

7.1.5 The Hand Tuning Method

In [42], Eggers and Jereniassen pursued a different method of measuring false sharing. They took various shared memory programs and modified them by hand to reduce false sharing. They ran the modified and unmodified versions of the programs, and defined the amount of false sharing to be the difference in execution time between the original and modified program. This method is valuable in that it does not rely on tracing techniques, but rather on directly measurable execution times of real programs. However, since hand-modification can vary in quality depending on the person making the modification and the complexity of the program, this definition fails the mathematic preciseness criterion.

7.1.6 The Cost Component Method

The cost model as presented in section 4.1 applies to a system with only a single block (one cache line or page), and is extended to the (normal) case wherein there is more than one block simply by treating each block as independent, and summing the costs for each block at the end of the simulation. However, for the purpose of quantifying false sharing, it is necessary to extend the model. For simplicity's sake, only machines without global memory or remote reference capability will be considered here. Thus, the only parameters that describe the machine are the block size and the remote block move cost, R. While in the model as presented in section 4.1 R is simply a model parameter without interpretation, in practice (as in Chapter 6) it will be a function of machine parameters such as block size, interprocessor latency and trap-handling speed. For the purpose of the false sharing analysis, assume

$$R = o + bs \tag{7.1}$$

where R is the remote block move model parameter, o is a constant representing overhead to set up a block move, b is a constant representing the bandwidth of the interconnection in terms of cost units per byte transferred, and s is the blocksize in words. That is, R is linear in the block size with a constant added. (The formulas provided for R in Chapter 6 have 2 for b, as seen in the pagesize/2 term[1], and $n\lambda$ plus o_h or o_s for o.)

When the blocksize is reduced, the coherence operations generated by the optimal policy will change. These changes will be due to one of three effects:

1. If data that are used together are separated, more coherence operations will be necessary to move them.
2. If falsely shared data are separated into pieces that are no longer falsely shared, coherence operations will be eliminated.
3. If a block is split into two pieces and only one of those pieces is used, the cost of moving the other piece will be saved.

Or, more concisely, overhead will increase for moving large blocks of data, false sharing will be reduced, and fragmentation will be reduced. The combination of these three effects results in the net change in cost between the initial and smaller blocksize models. The effect of false sharing at block size s is defined to be the difference in the value of the false sharing component between a run at block size s and a run with a single word block size. (One word block size machines thus have no false sharing).

Breaking up data that are used together ("useful groupings") results in increased cost because more operations are required to accomplish the same task. All of this additional cost will show up as additional overhead; the number of bytes transferred through the interconnection will not change. Reduction of false sharing reduces the total number of operations necessary, thus reducing both the number of bytes transferred and the amount of overhead incurred. Reducing fragmentation does not affect the number of operations, and so produces no change in the overhead, but reduces the volume of data transferred.

Given trace T and machine M with block size s with no global memory or remote reference, define the differences in grouping, fragmentation and false sharing between M and a version of M with block size one called M_1, to be Γ (gamma for grouping), F (F for fragmentation) and S (S for false Sharing), respectively. Define c_s and c_1 to be the optimal cost of T on M_s and M_1; ie., $c_s = c(\mathcal{O}, T, M_s)$ and $c_1 = c(\mathcal{O}, T, M_1)$. Then:

$$c_s - c_1 = \Gamma + F + S \tag{7.2}$$

[1] Recall that "pagesize" is in bytes, while s is in four byte words, thus leading to the difference in coefficients

Because breaking up groupings is detrimental, while decreasing false sharing and fragmentation are beneficial, Γ will be (zero or) negative, while F and S will be (zero or) positive.

Define Φ_s and Φ_1 to be the number of block moves in the optimal placements for T on M_s and M_1 respectively. Because M does not have remote reference or global memory, by the definition of cost from section 4.1.4 and by 7.1 above:

$$c_s = \sharp T + R_s\Phi_s = \sharp T + (o + bs)\Phi_s \tag{7.3}$$

$$c_1 = \sharp T + R_1\Phi_1 = \sharp T + (o + b)\Phi_1 \tag{7.4}$$

Combining 7.2, 7.3 and 7.4 yields:

$$\Gamma + F + S = c_s - c_1 = o(\Phi_s - \Phi_1) + b(s\Phi_s - \Phi_1) \tag{7.5}$$

The two terms on the right hand side of equation 7.5 are the *cost components*. $o(\Phi_s - \Phi_1)$ is the *overhead component* and $b(s\Phi_s - \Phi_1)$ is the *data transfer component*. (They may also be called the "latency" and "bandwidth" components, respectively.) Because of the reasoning above, Γ affects only the overhead component and F affects only the data transfer component. S affects both; define S_o and S_d to be the contribution of S to the change in overhead and in data transfer costs respectively. That is,

$$S_o = o(\Phi_s - \Phi_1) - \Gamma \tag{7.6}$$

$$S_d = b(s\Phi_s - \Phi_1) - F \tag{7.7}$$

and so by 7.5, 7.6 and 7.7:

$$S = S_o + S_d \tag{7.8}$$

If there were no fragmentation and no false sharing, and the block size were reduced from s to 1, then each transfer in M_s would become s transfers in M_1; that is, if $F = S = 0$, then $s\Phi_s = \Phi_1$. When there is false sharing, performance improvements show up as a number of transfers fewer than $s\Phi_s$; define τ to be the number of transfers saved by reduction of false sharing in M_1 as compared to M_s, so $0 \leq \tau \leq s\Phi_s$. If $F \neq 0$, then $\tau < s\Phi_s - \Phi_1$. τ is not directly measurable from the experiments, but rather is that fraction of transfers that were due to false sharing; knowing τ is equivalent to knowing F. Since the total effect of eliminating false sharing is the elimination of the transfers represented by τ, by the definition of S_o and S_d:

$$S_o = \frac{o\tau}{s} \tag{7.9}$$

and

$$S_d = b\tau \tag{7.10}$$

Combining 7.9 and 7.10 yields:

$$S_o = \frac{oS_d}{bs} \tag{7.11}$$

Together with 7.8, this produces:

$$S = \frac{oS_d}{bs} + S_d \tag{7.12}$$

7.12 and 7.7 combine to form:

$$S = (o + bs)\Phi_s - (\frac{o}{s} + b)\Phi_1 - (1 + \frac{o}{bs})F \tag{7.13}$$

Equation 7.13 provides a value for false sharing in terms of the measured numbers of transfers at block size s and 1 word, the model parameters and the cost savings due to reduction of fragmentation. This formula shouldn't be particularly surprising: the amount of false sharing is the total cost on M_s (excluding the cost of local references) of $(o + bs)\Phi_s$, minus the cost of transferring data in M_1 but in blocksize s chunks (thus $(\frac{o}{s} + b)\Phi_1$ rather than $(o + b)\Phi_1$); the additional overhead seen in M_1 is counted in Γ. The remaining term of 7.13 is a correction for reduction of fragmentation.

However, since there is no obvious way to measure the amount of fragmentation independent of false sharing, this expression can only be used to bound the amount of false sharing. Since F must never be less than zero, if F is set to zero then Equation 7.13 yields an upper bound on the amount of false sharing present in T on M_s:

$$S \leq (o + bs)\Phi_s - (\frac{o}{s} + b)\Phi_1 \tag{7.14}$$

The cost component definition of false sharing is fundamentally different than some of the other definitions, such as the full duration, interval or hueristic interval definitions. The cost component definition measures the total amount of performance improvement that could conceivably be had from elimination of false sharing, assuming that it were possible to do so without increasing overhead by using smaller block sizes, or merge operations. That is, F is what would be achieved if the system were able to move arbitrarily sized (and located) portions of blocks in single operations, moving only the data that is required for correctness. The only plausible way of doing such a thing is by directive from the application being run. Getting the kinds of savings shown by this method would require either very careful application tuning, or a very good compiler, library and/or runtime tools to assist the operating system in making these decisions.

Example Traces Showing Fragmentation, Grouping and False Sharing

The meaning of the terms defined in section 7.1.6 might easily be lost in all of the math. To illustrate their meanings, this section provides examples of synthetic traces that show lost grouping, decreased fragmentation and decreased false sharing, and then computes values for Γ, F and S for them. The synthetic traces used here demonstrate each of the effects in isolation, for increased clarity.

The first synthetic trace suffers from fragmentation, but not false sharing or loss of groupings. Consider trace T_F defined by:

$$T_F = (\mathbf{w}_0^{i \bmod p} | i \in 0..n)$$

where n is the length of the trace, $\sharp T = n$. This trace consists of a series of writes, cycling through by processor, to the first word of the page. That is, the list of references looks like:

$$T_F = (\mathbf{w}_0^0, \mathbf{w}_0^1, \mathbf{w}_0^2, \ldots, \mathbf{w}_0^0, \mathbf{w}_0^1, \ldots)$$

T_F has no false sharing: each word that is used at all is used by every processor. It loses no valuable groupings when the block size is reduced to one word, since only the first word of each block is ever used. However, it is severely fragmented, because each block move carries with it $s - 1$ words that are unused. Ignoring initial placement effects, every reference generates a block move in both M_s and M_1. Therefore, $\Phi_s = \Phi_1 = n$. So, the cost of T_F on M_s is $\sharp T + nR_s = n + no + nbs$ and on M_1 it is $\sharp T + nR_1 = n + no + nb$; the difference in cost is $c_s - c_1 = nb(s-1)$. All of this cost difference is due to reduction of fragmentation, so $F = nb(s-1)$. Substituting these values into equation 7.13 yields, as one would expect, $S = 0$; further substituting into 7.2 shows that $\Gamma = 0$.

The next synthetic trace, T_S, exhibits false sharing, but no change in grouping or fragmentation:

$$T_S = (\mathbf{w}_{i \bmod s}^{i \bmod p} | i \in 0..n)$$

Where s is block size, here assumed to be evenly divisible by p, the number of processors. This trace cycles through processors, each writing the next word sequentially. Thus, again ignoring initial placement effects, each reference results in a block move, but no information is communicated between processors. There is no fragmentation, because all of the block moves result from false sharing, and there are no useful groupings lost, since there are no useful groupings. Since each reference results in a page move, $\Phi_s = n$. In M_1, however, since no data is communicated, no blocks are moved, and so $\Phi_1 = 0$. Thus, $c_s = n + n(o + bs)$, $c_1 = n$ and $c_s - c_1 = n(o + bs)$. Because there is no fragmentation, $F = 0$. Substituting these values into equation 7.13 results in $S = (o + bs)\Phi_s$ and so by 7.2, $\Gamma = 0$.

The third and final example trace, T_Γ exhibits a reduction in useful grouping, while not changing false sharing or fragmentation.

$$T_\Gamma = (\mathbf{w}_{i \bmod s}^{\lfloor \frac{i}{s} \rfloor} | i \in 0..n)$$

Here $\lfloor x \rfloor$ represents the floor function of x: the greatest integer less than or equal to x. This trace has processor 0 write each word of the block, followed by processor 1 writing each word of the block, and so on until each processor has written each word of the block. Then, the process repeats. Ignoring initial placement costs, and assuming that the length of the trace is such that it ends with the last processor making its last write to the block, we can compute the cost of execution on M_s and M_1. On M_s, the block is moved once per iteration for each processor, so $\Phi_s = \frac{n}{s}$. This is also true on M_1, but each block is only one word long, so $\Phi_1 = n$. In this instance, it is not necessary to make assumptions about F, because the value of the two terms in equation 7.13 that do not depend on F is 0, and the coefficient on the F term is negative. Therefore, either $F < 0$, $S < 0$ or $F = S = 0$. Since there cannot be negative false sharing or fragmentation, we conclude that S and F are zero. Therefore, by 7.2 $\Gamma = c_s - c_1 = \frac{(1-s)no}{s}$. Recall that Γ is typically negative.

7.2 Estimating False Sharing

None of the definitions discussed in section 7.1 provide a way of measuring false sharing. Equation 7.13 from the cost component method could be used to show false sharing as a function of F, the level of fragmentation. However, the possible range of false sharing thus demonstrated is very large: for most applications, the contribution of false sharing to cost could range from nothing to over 90%.

Figures 7.2, 7.3, 7.4 and 7.5 show the cost of execution of the CC and NUMA models broken down into overhead and data transfer components for `plytrace`, `p-qsort`, `cholesky` and `sorbyr` respectively. While it is not possible to determine the amount of false sharing directly from these graphs, looking at the degree to which the breakdowns vary over the range of block sizes gives a hint.

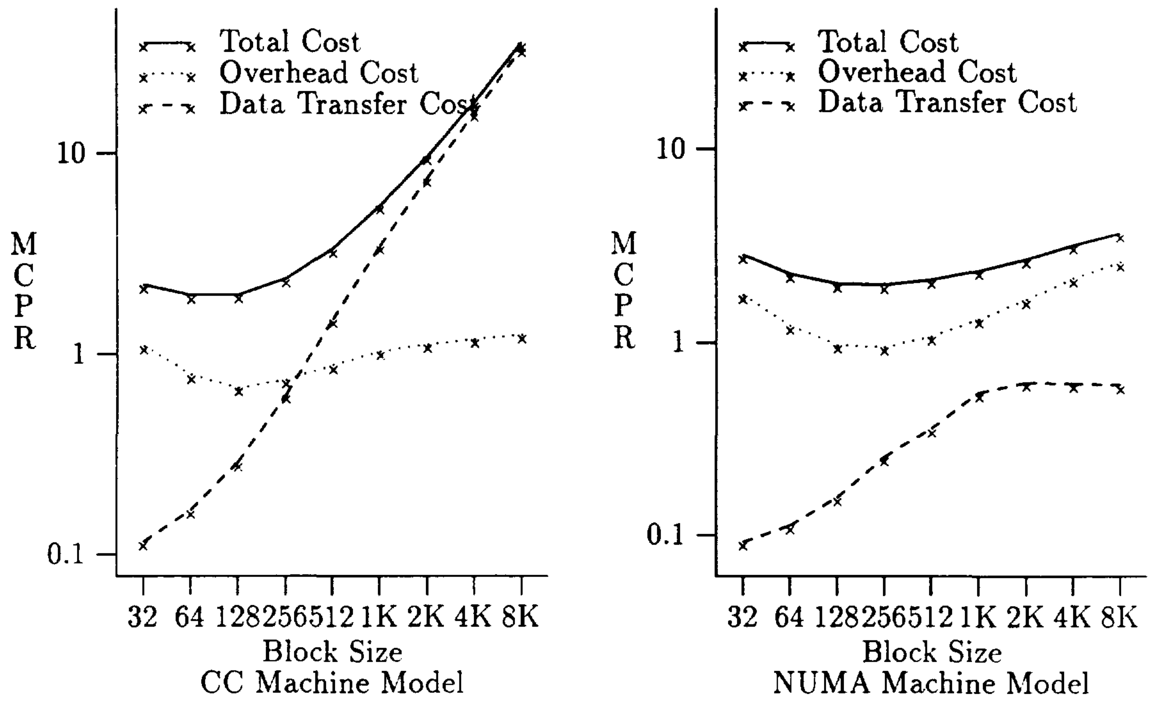

Figure 7.2: **plytrace** MCPR broken down into data transfer and overhead components (log scales)

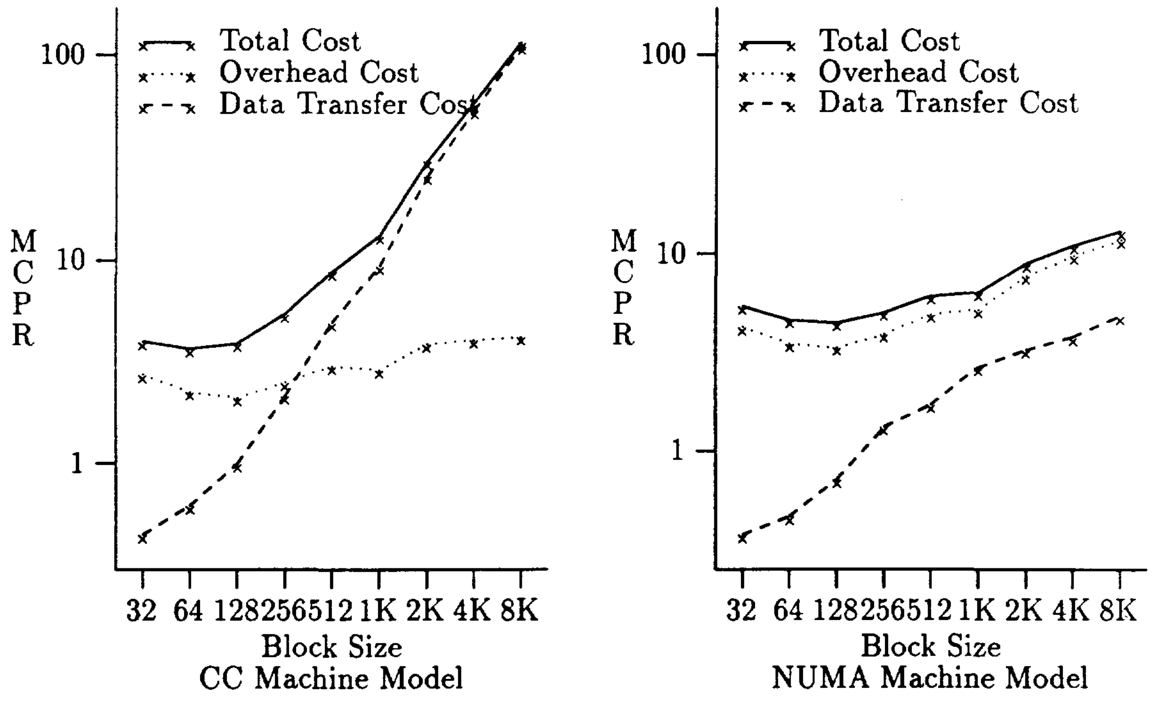

Figure 7.3: **p-qsort** MCPR broken down into data transfer and overhead components (log scales)

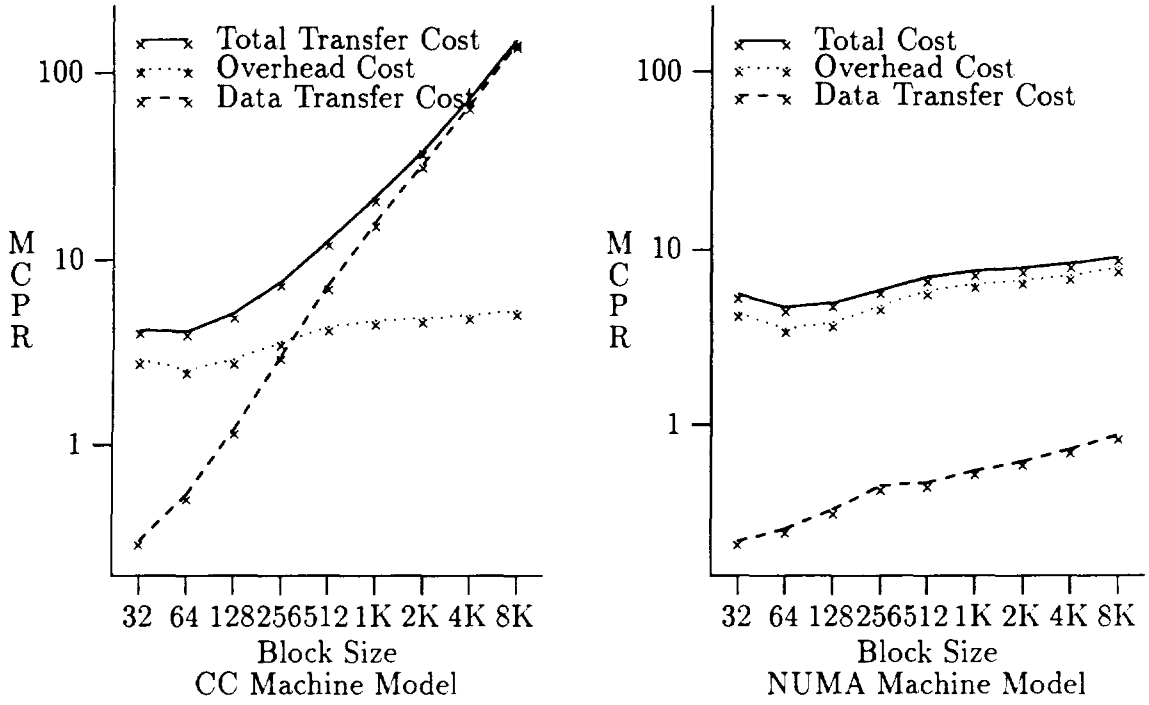

Figure 7.4: `cholesky` MCPR broken down into data transfer and overhead components (log scales)

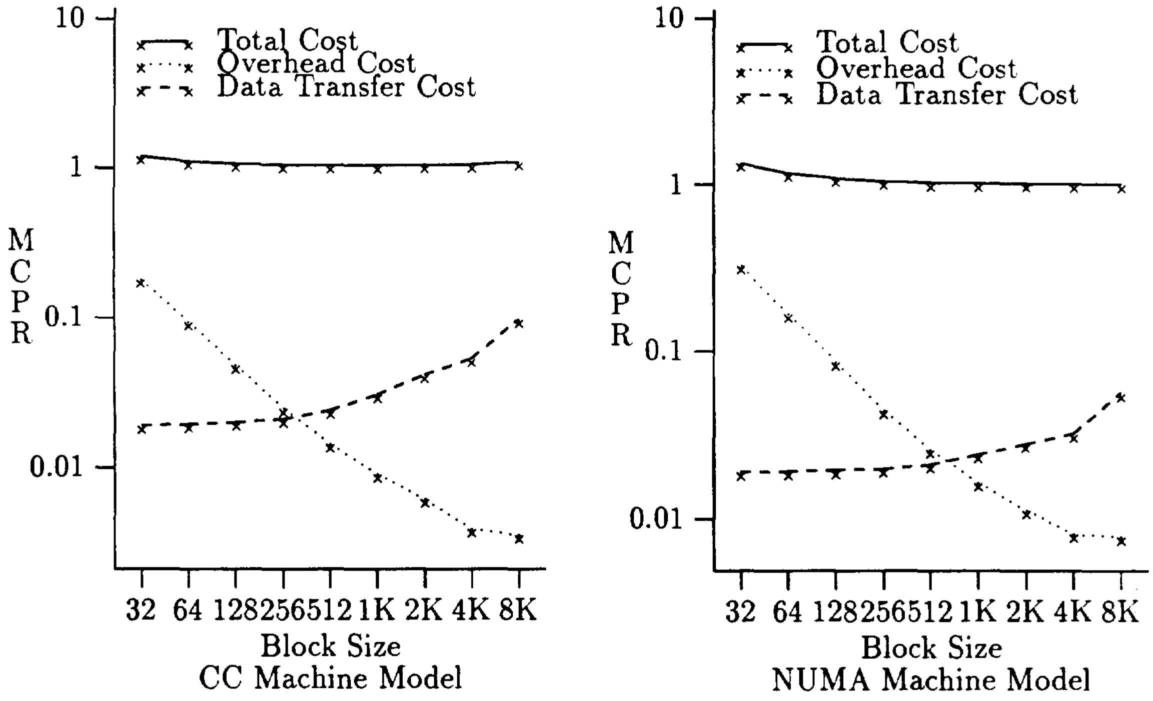

Figure 7.5: `sorbyr` MCPR broken down into data transfer and overhead components (log scales)

Γ, the "useful grouping" cost component, can only result in increased cost with reduced block sizes. Fragmentation does not affect the overhead component at all. Therefore, any reduction in overhead must be due to a reduction in false sharing. Furthermore, that reduction in overhead must be accompanied by some reduction in data transfer as well, because the reduction in false sharing results in fewer total transfers needed, and a transfer incurs both overhead and data transfer costs, as indicated by equation 7.11.

A bound can be placed on the possible change in fragmentation when the block size is halved. When the block size is halved, fragmentation can at most be reduced by a factor of two. Let an arbitrary multiword block of size s, machine M_s and trace T that references only the one block be given. Assume that there is no false sharing in T. Define $S \subseteq T$ to be the set of references that result in block moves in T. Consider running T on machine $M_{\frac{s}{2}}$, which differs from M_s in that is has half the block size. Since there is no false sharing in T, any reference in S represents a real data communication. Since the blocks in $M_{\frac{s}{2}}$ are sub-blocks of those in M_s, any transfer in $M_{\frac{s}{2}}$ can not move any data that wouldn't have also been moved in M_s. Therefore, every reference in S results in a transfer in T on $M_{\frac{s}{2}}$. Since the data transfer portion of R in $M_{\frac{s}{2}}$ is half that of M_s, the data transfer cost of T on $M_{\frac{s}{2}}$ is at least half of that of T in M_s. (It could be more, because some references in $T \setminus S$ could also initiate block moves in $M_{\frac{s}{2}}$.) Since any change in fragmentation shows up only in the data transfer component, we conclude that when the block size is halved, reduction in fragmentation can account for at most a halving of the data transfer cost component.

Iterating this result shows that when block size changes from s to $\frac{s}{2^n}$, fragmentation can reduce the data transfer cost component by at most a factor of 2^n. (The result could be generalized to denominators that are not powers of two, but that is unnecessary for this analysis.) Many of the applications approach or even exceed this limit over a wide range of block sizes. For example, the data transfer cost component for **p-qsort** with 8K blocks on the CC model is 1.2 billion cost units. At a 1K block size the data transfer cost is 138 million, for a ratio of 8.7, which is greater than the factor of 8 that can be explained by fragmentation alone. At a 128 byte block size the data transfer cost is 28 million cost units, or 43 times less than that at 8K; while this could be explainable entirely by fragmentation, doing so would mean that nearly all of the memory in the 8K blocks was unused. This is unlikely. False sharing is probably responsible for most of the difference in performance

On the other hand, **sorbyr**, as shown in figure 7.5, is an application that has very little false sharing. The data transfer cost component for **sorbyr** is never greater than 10% of the total cost of running the application (recall that the total cost as shown in these graphs is the sum of not only the data transfer and overhead components, but also the cost of making local memory references, which contributes 1 to the MCPR). The ratio of the data transfer cost at an 8K block size to that at a 32 byte block size is 5 to 1, which is much less than the factor of 256 that could be accounted for by fragmentation. The overhead component steadily increases in its contribution to cost as the block size is reduced, in stark contrast to the other applications discussed. Graphs like that of `sorbyr` are typical of applications having little false sharing.

Most of the applications in the application suite display performance like that of `p-qsort`, `plytrace`, or `cholesky` rather than like `sorbyr`. This indicates that false sharing is probably the major reason for the poor performance of these applications on large block size machines. Thus, if false sharing is somehow reduced large block size machines could be expected to perform comparable to small block size machines.

8 Conclusions

The primary conclusion of this work is:

- Properly designed software-coherent shared-memory multiprocessors can perform competitively with coherently-cached machines.

This conclusion is based on the direct evidence offered by the simulations in Chapter 6. These simulations show that NUMA machines with small pages have performance that is nearly as good as that of a coherently cached-machine built of similar speed components. In particular, the NUMA machine model performs within 20% of CC on all but one of the non-Presto applications in the application set; the performance of the Presto applications is sufficiently bad that they would probably not be run unmodified on any machines with remote latencies of this magnitude.

The conclusion is supported by several main points, to wit:

- The tracing process created memory reference traces that do not differ from what would happen in reality enough to significantly distort final results.
- The cost model captures the essential features of NUMA and coherently-cached systems well enough to describe first order performance effects.
- The optimal analysis technique eliminates bias that might be present if on-line policies were used, and so allows the exploration of the limits of the hardware without consideration of the policies being used.
- At least for some machines, on-line policies can be created that approach optimal performance for real applications, and so optimal behavior is not completely unrealistic as a guide for overall machine performance.
- The machine models used for the final comparison are based on components of speed comparable to that of modern hardware.

- The construction of kernel software to support NUMA systems is not terribly difficult, and should not be seen as a major impediment to their construction.

In addition to the work that directly supports the thesis, this dissertation also makes the following contributions:

- An implementation of a NUMA kernel within the pmap layer of Mach that showed that Mach's virtual memory system was well designed to support novel architectures such as NUMAs with only minor modifications to the machine independent layer.
- A formal model of shared memory multiprocessors and program execution on them that is amenable to optimal analysis and also allows mathematical proof of properties of machines and traces.
- A demonstration that NUMA policies need to be tuned to the hardware on which they are run.
- Showing that the value of remote reference depends on the size of the block being used. For the block sizes typically employed in paged machines, it can yield significant performance improvements. Well coded remote reference software is a promising option for DSM machines, and merits experimental implementation, similar to Spector's work on the Alto [89, 90], but specifically tuned for remote memory reference. Conversely, remote reference is unlikely to benefit cache-coherent machines enough to warrant the expense of building it, at least when the line size is small.
- Finding that distributed shared memory systems with large page sizes are not competitive as a base for generic shared-memory programs. To properly exploit them, techniques such as those used in the Munin system should be employed, or programmers should keep data migrations in mind when writing programs for them.

Furthermore, the work in Chapters 6 and 7 indicates that the primary reason that NUMA machines perform better with smaller page sizes is that reducing the page size also reduces false sharing, and in systems with very large interprocessor communication latencies such false sharing is extremely expensive.

8.1 Future Work

There are many directions in which this work could be extended:

- Exploring ways of reducing false sharing in applications, by using complier, run-time, or programming language tools, or by assisting the programmer in eliminating it by hand.
- Considering the effects of staticly or dynamically variable sized blocks, such as suggested by Dubnicki for cache lines [41].
- Reducing the error in the studies by doing more detailed simulation, including effects such as contention and feedback affecting instruction ordering.
- Building an implementation of a software coherent system as described here.
- Finding better ways to define and quantify the effects of false sharing.
- Considering machines with properties disallowed by the current model, such as weak coherence, latency tolerance or microtasking.
- Extending the technique of optimal analysis to areas other than NUMA policy design.

8.1.1 Reducing False Sharing

The work in chapter 7 suggests that the main reason that shared memory machines perform better with smaller block sizes is the reduction of false sharing with block size. If it is possible to reduce false sharing by program restructuring, then the main advantage of smaller block sizes would be lessened. The decision about what block size to choose would involve weighing fragmentation against overhead costs; this would probably result in machines with large block sizes.

Given that false sharing is so elusive to measure, removing it will likely be difficult, too. The most appealing method is to have language, runtime and compiler tools that do the work for the programmer, allowing UMA-style programs to run well on NUMA, where performance degradation comes only from real communication.

If tools to automatically reduce false sharing could not be created, then it might be possible to provide hints to the programmer about changes that might be made to increase program speed. A tool of this sort could operate by instrumenting the program and examining its real communications behavior at runtime. When it detected communications events that did not correspond to real communication, it could determine which portions of program memory caused this behavior and suggest corrective action.

8.1.2 Variable Sized Blocks

Dubnicki [41] suggests hardware that supports dynamically variable block sizes. This approach is promising in that it could allow systems to adapt to programs that have both small and large size migratory data. This would reduce false sharing, without incurring the greatly increased overhead due to exclusively using small blocks.

8.1.3 Increasing the Detail of the Model

The model used in this work assumes infinite memories and caches, ignores contention in the interconnection and memory systems, assumes that program behavior is independent of page movement decisions, treats TLB effects as an afterthought, and ignores the effects on higher level virtual memory systems of having very small pages. All of these points introduce errors of one sort or another into the performance estimates provided by the model.

To correct for most of these types of problems would require switching from a mathematical model to a detailed architectural simulation. This would also require at best a rethinking of the optimal NUMA policy technique, and at worst complete abandonment of it, depending on the complexity of the new simulation. Nevertheless, such a simulation would be necessary before undertaking an implementation of such a machine.

8.1.4 More Complicated Machines

Another weakness of the model is that it is unable to deal with more complicated machine models, such as those with different coherence models, latency tolerance, and more complicated interprocessor interconnections. To simulate weaker coherence would require new traces that included information about not only synchronization points, but also which objects are protected by which synchronization operations, an extension of the cost model similar to that proposed in section 7.1.2 that includes a cost for merging pages, and a revision of the optimal placement algorithm.

To simulate a multi-threaded latency-tolerant machine would require traces describing an application designed for such a machine, and a model of the context switching behavior of the machine. Of course, optimal placement is less of an issue, but one must decide how many threads per processor to create, and how to schedule those threads.

Simulating a more complicated interconnection network is more straightforward given the mechanics developed in this dissertation. All that would be required would be a good formal model of the cost of the various inter-memory operations and a generalization of the optimal placement algorithm. The first step in this generalization has already been taken by the global memory version of optimal placement.

8.1.5 Other Uses of Optimal Analysis

Optimal analysis can potentially be a fruitful technique for exploration of areas other than NUMA memory management. Any area of computer science in which an algorithm uses a hueristic to attempt to minimize some value without future knowledge is a potential target for optimal analysis. For example, virtual memory page replacement algorithms, cache replacement algorithms, or the code in a compiler that selects which registers to spill could all potentially benefit from optimal analysis. To apply it requires solving an optimization problem that may or may not be NP-hard; whether it is depends on the particular problem at hand. I believe that optimal analysis can be a fruitful technique for exploring performance of a wide class of computer systems problems.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX*, July 1986.

[2] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 2–14, 1990.

[3] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 104–114, 1990.

[4] A. Agarwal, R. Simoni, J. Hennessey, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual Symposium on Computer Archetecture*, pages 280–289, May 1988.

[5] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 119–125, 1986.

[6] R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: A Semi-Automatic Parallel Programming Assistant. In *Proceedings of the International Conference on Parallel Processing*, pages 164–170, 1986.

[7] R. J. Anderson. An Experimental Study of Parallel Merge Sort. Technical Report 88-05-01, University of Washington Department of Computer Science, May 1988.

[8] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[9] Apollo, Inc. *The Apollo DOMAIN Architecture.* Apollo Computer, Inc., Chelmsford, MA, 1981.

[10] J. Archibald and J.-L. Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 355–362, June 1984.

[11] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[12] AT&T. *System V Interface Definition.* Computer Information Center, PO Box 19901, Indianapolis, IN, 1986.

[13] S. J. Baylor and B. D. Rathi. An Evaluation of Memory Reference Behavior of Engineering/Scientific Applications in Parallel Systems. Technical Report 14287, IBM, June 1989.

[14] BBN. *Butterfly Parallel Processor Overview.* BBN Laboratories, BBN Report 6148, Cambridge, Massachusetts, March 1986.

[15] BBN. *Inside the Butterfly Plus.* BBN Advanced Computers, Cambridge, MA, October 1987.

[16] B. Beck and D. Olien. A Parallel Programming Process Model. In *Proceedings of the Winter USENIX Conference*, 1987.

[17] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Arichtectures. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 125–134, 1990.

[18] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 125–134, 1990.

[19] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Symposium on Principles and Practice of Parallel Programming, SIGPLAN Notices 25(3)*, pages 168–176, March 1990.

[20] M. Berry, et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. CSRD Report 827, Center for Supercomputing Research and Development, University of Illinois, May 1989.

[21] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[22] D. Black, A. Gupta, and W.-D. Weber. Competitive Management of Distributed Shared Memory. In *Proceedings, Spring COMPCON*, pages 184–190, February 1989.

[23] D. L. Black and D. D. Sleator. Competitive Algorithms for Replication and Migration Problems. Technical report, Carnegie-Mellon University, Computer Science Department, November 1989. CMU-CS-89-201.

[24] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[25] W. J. Bolosky and M. L. Scott. A Trace-Based Comparison of Shared Memory Multiprocessors using Optimal Off-Line Analysis. Technical Report 432, University of Rochester Computer Science Department, December 1991.

[26] W. J. Bolosky and M. L. Scott. Evaluation of Multiprocessor Memory Systems Using Off-Line Optimal Behavior. *The Journal of Parallel and Distributed Computing*, August 1992.

[27] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, 1991.

[28] E. Brooks. BBN TC2000 Architecture and Programming Model. In *Proceedings of the IEEE COMPCON*, February 1991.

[29] N. Carriero and D. Gelernter. Applications Experience with Linda. In *Proceedings, Parallel Programming: Experience with Applications, Languages and Systems*, pages 173–187, July 1988.

[30] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 152–164, 1991.

[31] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[32] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[33] E. Chaves, P. Das, T. LeBlanc, B. Marsh, and M. Scott. Kernel-Kernel Communication in a Shared-Memory Multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, May 1993.

[34] D. R. Cheriton, H. A. Goosen, and P. D. Boyle. Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture. *Computer*, 24(2):33–46, February 1991.

[35] D. R. Cheriton, H. A. Goosen, and P. Machanick. Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 109–118, Tokyo, April 1991.

[36] D. W. Clark. Cache Performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, February 1983.

[37] E. Cooper and R. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, Computer Science Department, March 1987.

[38] A. L. Cox. *The Implementation and Evaluation of a Coherent Memory Abstraction for NUMA Multiprocessors.* Ph.D. thesis, University of Rochester, 1992.

[39] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[40] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.

[41] C. Dubnicki and T. J. Leblanc. Adjustable Block Size Coherent Caches. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 170–180, May 1992.

[42] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I-377–I-381, 1991.

[43] S. J. Eggers and R. H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the 3rd International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 257–270, April 1989.

[44] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *Proceedings*

of the 1990 SIGMETRICS Conference on Measurement and Modelling of Computer Systems, pages 37–47, May 1990.

[45] B. D. Fleisch and G. J.Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[46] A. Forin, J. Barrera, and R. Sanzi. The Shared Memory Server. In *Proceedings of the Winter USENIX*, pages 229–244, Jan 1989.

[47] R. J. Fowler. Personal communication. Email commenting on PLATINUM applications, April 1993.

[48] R. J. Fowler and A. L. Cox. An Overview of PLATINUM: A Platform for Investigating Non-Uniform Memory. Technical Report TR-262, Computer Science Department, University of Rochester, Nov. 1988.

[49] A. Garcia. *Efficient Rendering of Synthetic Images.* Ph.D. thesis, Massachusetts Institute of Technology, February 1988.

[50] A. Garcia, D. Foster, and R. Freitas. The Advanced Computing Environment Multiprocessor Workstation. Research Report RC-14419, IBM T.J. Watson Research Center, March 1989.

[51] M. Gardner. Mathematical Games. *Scientific American*, 223(4):120–123, October 1970.

[52] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, 1990.

[53] S. Goldschmidt and H. Davis. Tango Introduction and Tutorial. Technical Report CSL-TR-90-410, Computer Systems Laboratory, Stanford University, Jan 1990.

[54] J. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[55] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, 1992.

[56] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[57] M. A. Holliday. On the Effectiveness of Dynamic Page Placement. Technical Report CS-1989-19, Department of Computer Science, Duke University, September 1989.

[58] M. A. Holliday. Reference History, Page Size, and Migration Daemons in Local/Remote Architectures. In *Proceedings of the 3rd International Conference on Architectural Support Support for Programming Languages and Operating Systems*, April 1989.

[59] IBM. *IBM RT/PC Hardware Technical Reference*. IBM, 1988. Part Numbers SA23-2610-0, SA23-2611-0 and SA23-2612-0, Third Edition.

[60] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[61] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall Software Series, Englewood Cliffs, New Jersey, 1978.

[62] E. J. Koldinger, S. J. Eggers, and H. M. Levy. On the Validity of Trace-Driven Simulations for Multiprocessors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 244–253, 1991.

[63] R. P. LaRowe and C. S. Ellis. Virtual Page Placement Policies for NUMA Multiprocessors. Technical report, Department of Computer Science, Duke University, December 1988.

[64] R. P. LaRowe and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[65] R. P. LaRowe, C. S. Ellis, and L. S. Kaplan. The Robustness of NUMA Memory Management. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 137–151, 1991.

[66] J. R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software: Practice and Experience*, 20(12):1241–1258, December 1990.

[67] T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Memory Management for Large-Scale NUMA Multiprocessors. Technical Report 311, University of Rochester Computer Science Department, 1989.

[68] R. B. Lee. Precision Architecture. *Computer*, 22(1):78–91, January 1989.

[69] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, 1990.

[70] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, 1992.

[71] K. Li. IVY: A Shared Memory Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages II–94–II–101, 1988.

[72] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[73] K. Li and R. Schaefer. A Hypercube Shared Virtual Memory System. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–125–I–132, 1989.

[74] E. Markatos. *Scheduling for Locality in Shared-Memory Multiprocessors.* Ph.D. thesis, University of Rochester, Department of Computer Science, 1988.

[75] E. P. Markatos and T. J. LeBlanc. Using Memory (or Cache) Affinity in Loop Scheduling on Shared-Memory Multiprocessors. Technical Report 410, Computer Science Department, University of Rochester, March 1992. To appear, Supercomputing '93 or IEEE Transactions on Parallel and Distributed Systems.

[76] E. M. McCreight. The Dragon Computer System, an Early Overview. In *NATO Advanced Study Insitiute on Microarchitecture of VLSI Computers*, July 1984.

[77] J. M. Mellor-Crummey and M. L. Scott. Synchronization without Contention. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, 1991.

[78] M. Misra, editor. *RISC System/6000 Technology.* IBM, Austin, TX, 1990. Book SA23-2619.

[79] H. E. Mizrahi, J.-L. Baer, E. D. Lazowska, and J. Zahorjan. Introducing Memory into the Switch Elements of Multiprocessor Interconnection

Networks. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 158–166, May 1989.

[80] Motorola. *MC68881/MC68882 Floating-Point Coprocessor User's Manual.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[81] J. M. Ortega and R. G. Voigt. Solution of Partial Differential Equations on Vector and Parallel Computers. *SIAM Review*, 27(2):149–240, June 1985.

[82] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[83] U. Ramachandran, M. Ahamad, and M. Y. A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Proceedings of the International Conference on Parallel Programming*, pages II–160–II–169, June 1989.

[84] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[85] E. Rothberg and A. Gupta. Techniques for Improving the Performance of Sparse Factorization on Multiprocessor Workstations. In *Proceedings, Supercomputing '90*, November 1990.

[86] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[87] A. D. Samples. Mache: No-Loss Trace Compaction. *Performance Evaluation Review*, 17(1):89–97, May 1989.

[88] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[89] A. Z. Spector. *Multiprocessor Architectures for Local Computer Networks.* Ph.D. thesis, Department of Computer Science, Stanford University, August 1981.

[90] A. Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):246–260, April 1982.

[91] P. Stenstrom. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, June 1990.

[92] J. Stone and A. Norton. *The VM/EPEX FORTRAN Preprocessor Reference*. IBM, 1985. Research Report RC11408.

[93] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *Computer*, 23(5):54–64, May 1990.

[94] C. B. Stunkel and W. K. Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. *Performance Evaluation Review*, 17(1):70–78, May 1989.

[95] C. P. Thacker and L. C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of the 2nd International Conference on Architectural Support Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[96] S. Thakkar, P. Gifford, and G. Fielland. Balance: A Shared Memory Multiprocessor. In *Proceedings of the 2nd International Conference on Supercomputing*, May 1987.

[97] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Software TLB Management in OSF/1 and Mach 3.0. In *Proceedings of the 3rd USENIX Mach Symposium*, April 1993.

[98] J. E. Veenstra and R. J. Fowler. A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 149–161, 1992.

[99] Z. G. Vranesic, M. Stumm, D. M. Lewis, and R. White. Hector: A Hierarchically Structured Shared-Memory Multiprocessor. *Computer*, 24(1):72–79, January 1991.

[100] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support Support for Programming Languages and Operating Systems*, April 1989.

[101] B. Wheeler and B. Bershad. Consistency Management for Virtually Indexed Caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 124–136, 1992.

[102] P. C. Yew. Architecture of the Cedar Parallel Supercomputer. Technical Report 609, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, August 1986.

[103] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, 1987.

Appendix A Results for All Applications

This appendix contains graphs that show the effects of page size and software overhead for those applications that are not used as examples in Chapter 6. They are included for completeness' sake.

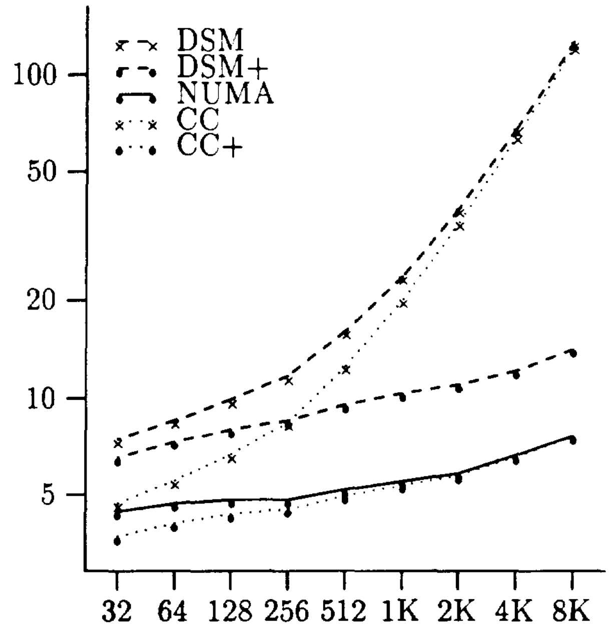

Figure A.1: **e-hyd** MCPR vs. block size (log scales)

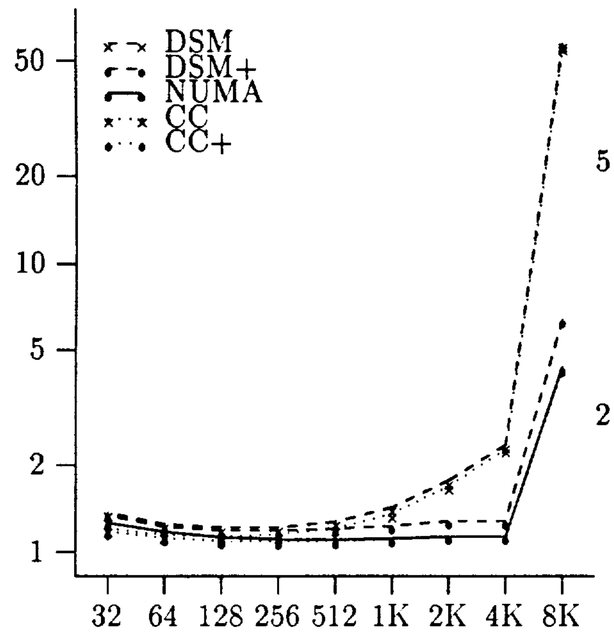

Figure A.2: **gauss** MCPR vs. block size (log scales)

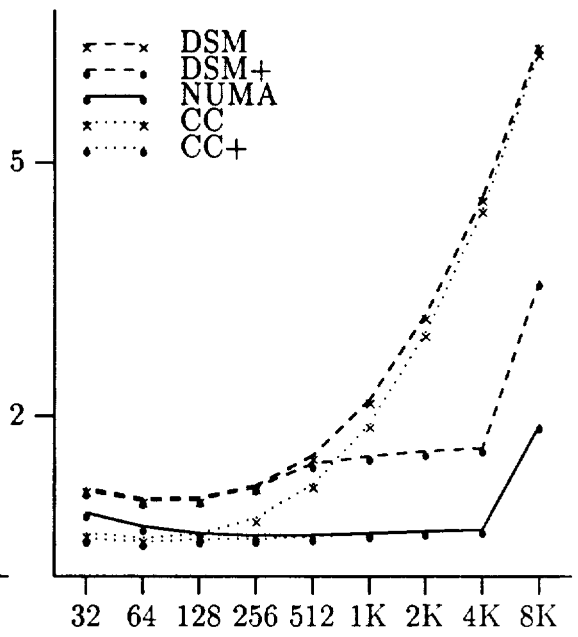

Figure A.4: **sorbyc** MCPR vs. block size (log scales)

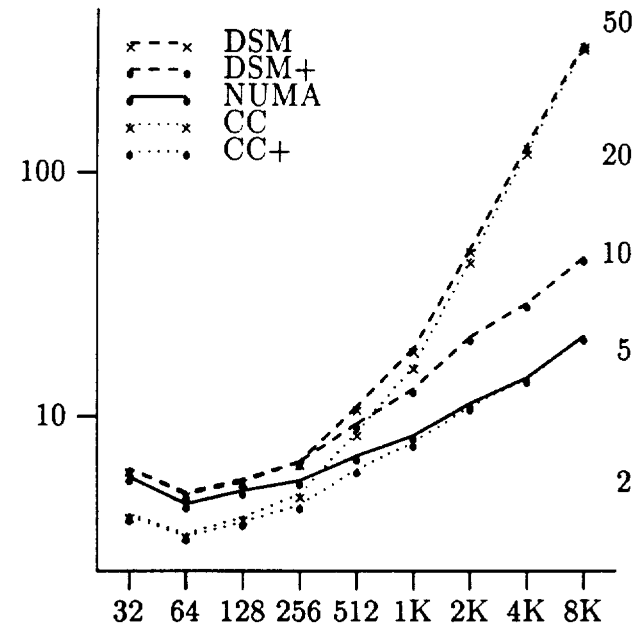

Figure A.3: **p-life** MCPR vs. block size (log scales)

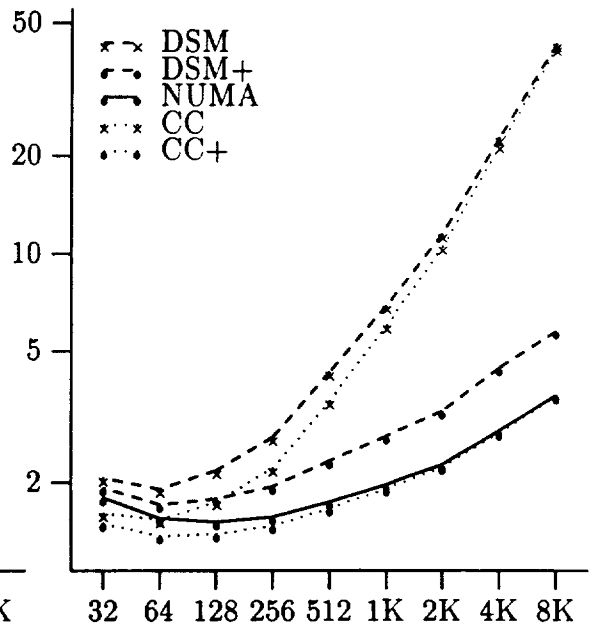

Figure A.5: **matmult** MCPR vs. block size (log scales)

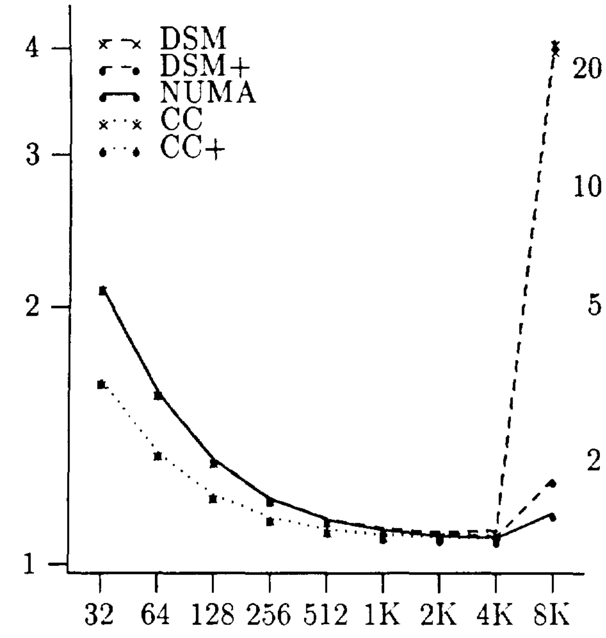

Figure A.6: `bsort` MCPR vs. block size (log scales)

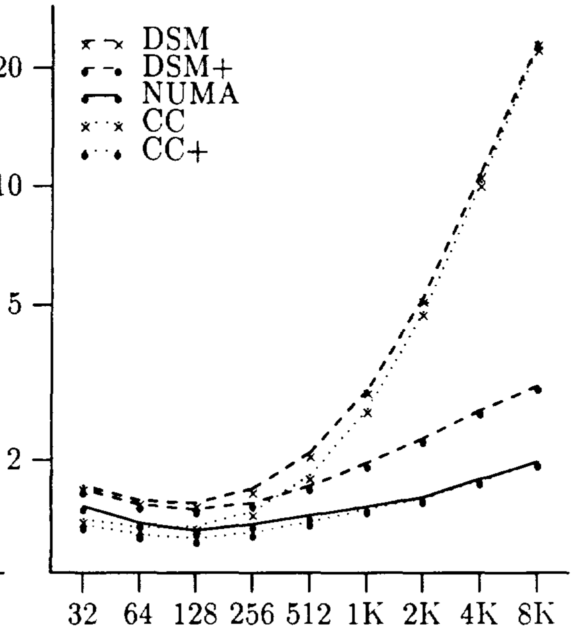

Figure A.8: `e-nasa` MCPR vs. block size (log scales)

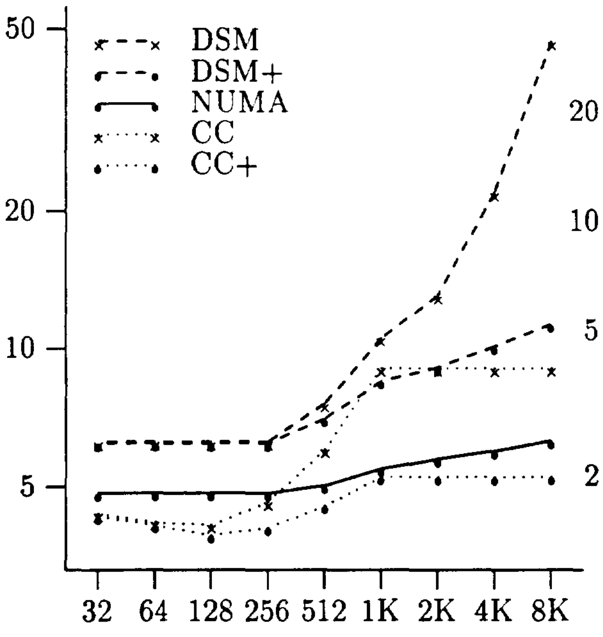

Figure A.7: `e-simp` MCPR vs. block size (log scales)

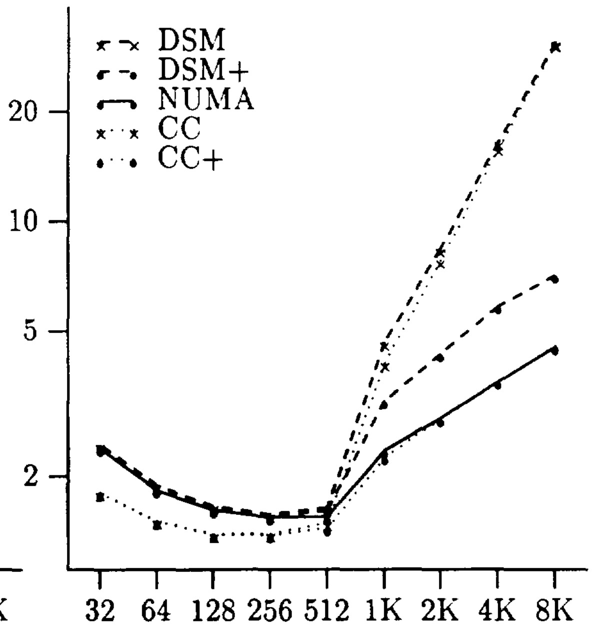

Figure A.9: `p-matmult` MCPR vs. block size (log scales)

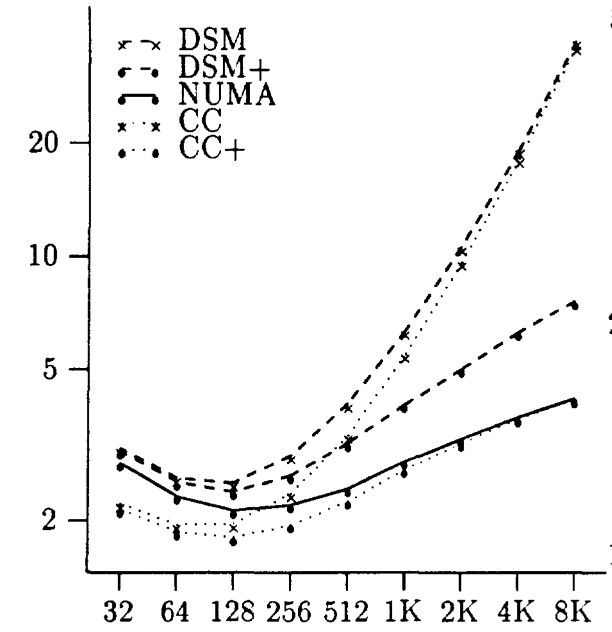

Figure A.10: `plytrace` MCPR vs. block size (log scales)

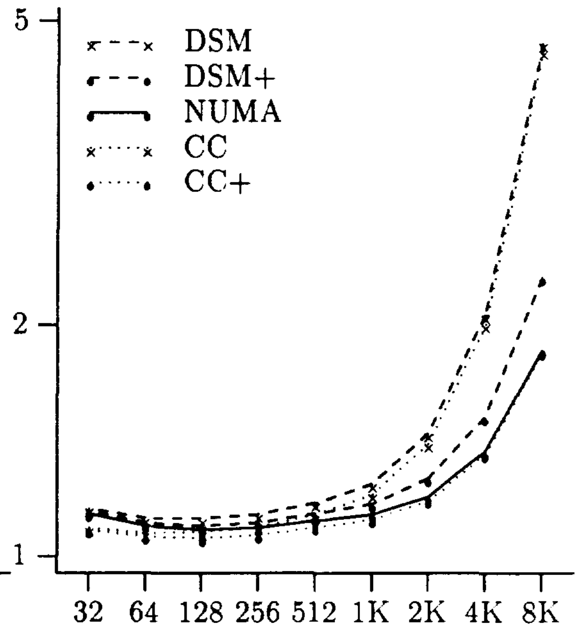

Figure A.12: `water` MCPR vs. block size (log scales)

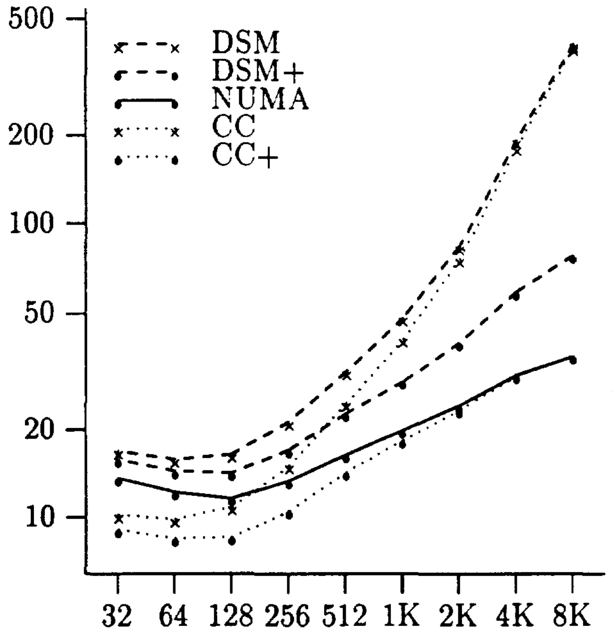

Figure A.11: `mp3d` MCPR vs. block size (log scales)

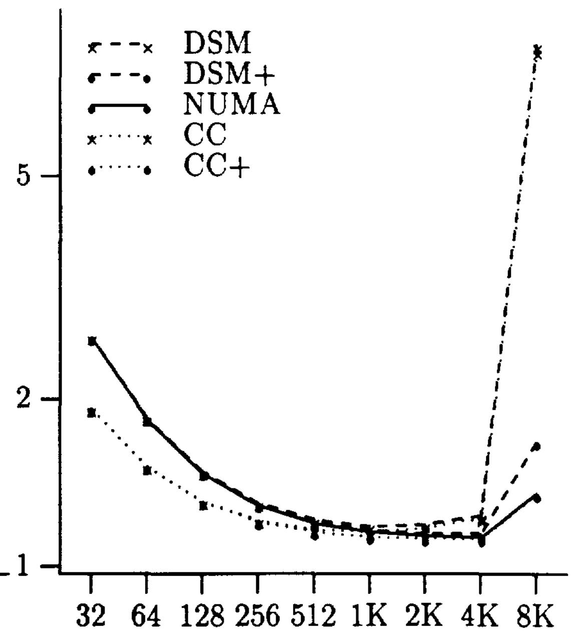

Figure A.13: `kmerge` MCPR vs. block size (log scales)

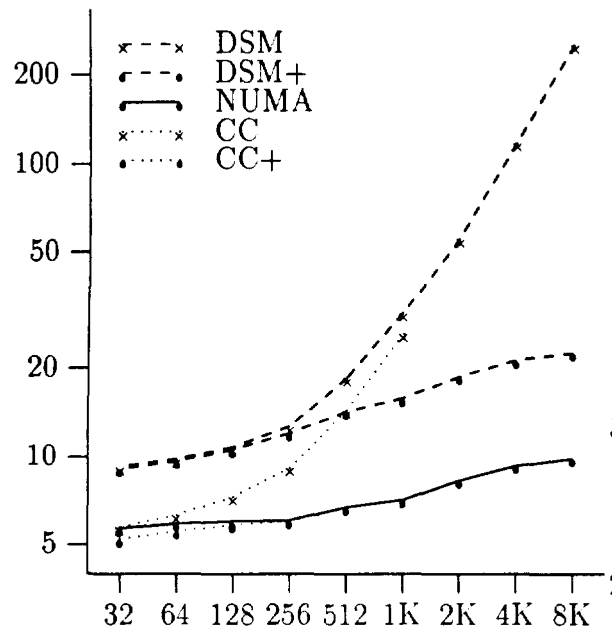

Figure A.14: fft MCPR vs. block size (log scales)

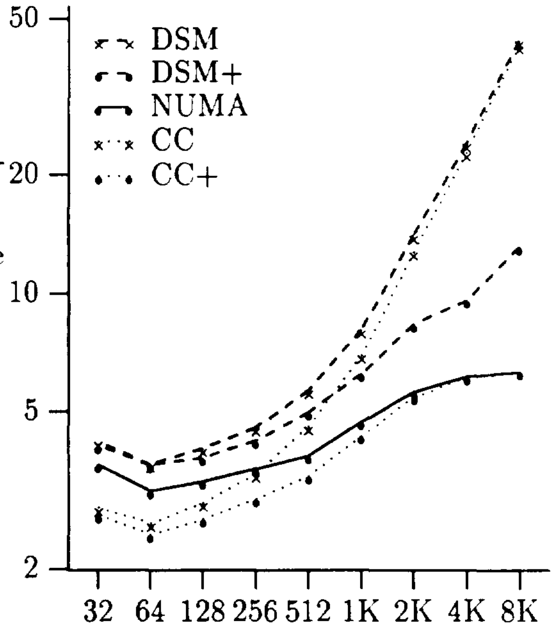

Figure A.16: chip MCPR vs. block size (log scales)

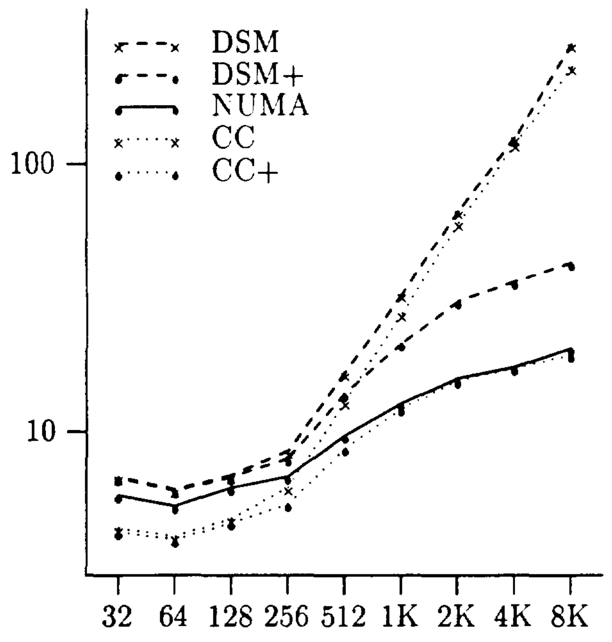

Figure A.15: p-gauss MCPR vs. block size (log scales)

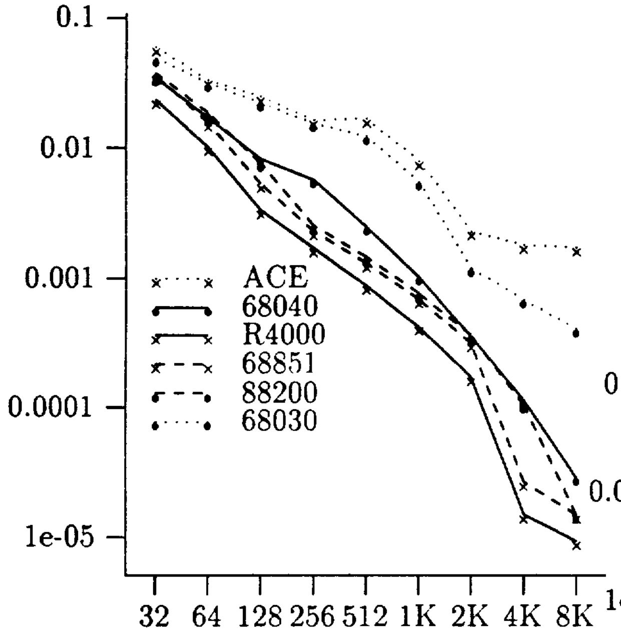

Figure A.17: `p-qsort` TLB miss rate vs. page size (log scales)

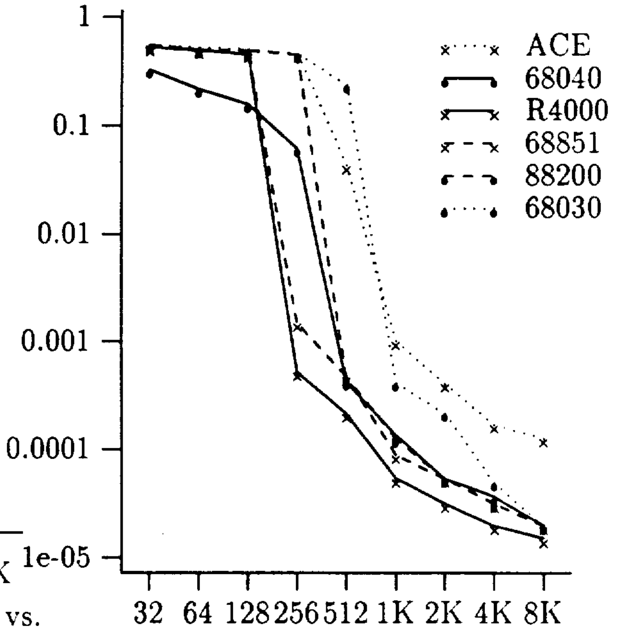

Figure A.18: `matmult` TLB miss rate vs. page size (log scales)

Figures A.17 through A.29 show tlb performance as a function of pagesize.

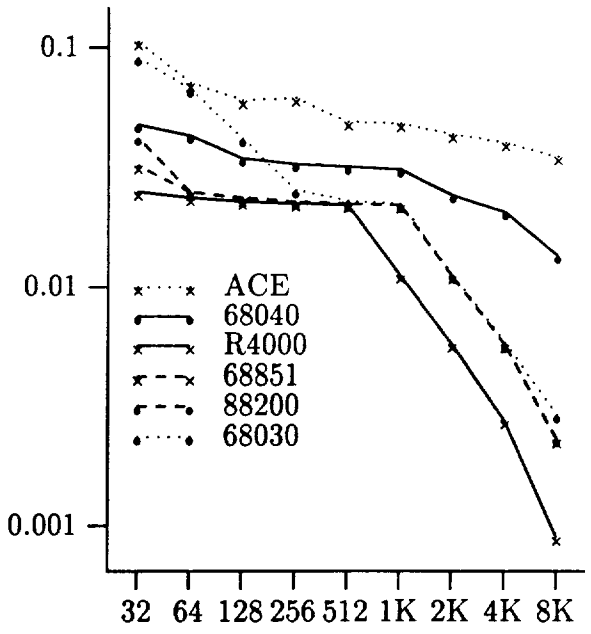

Figure A.19: `e-simp` TLB miss rate vs. page size (log scales)

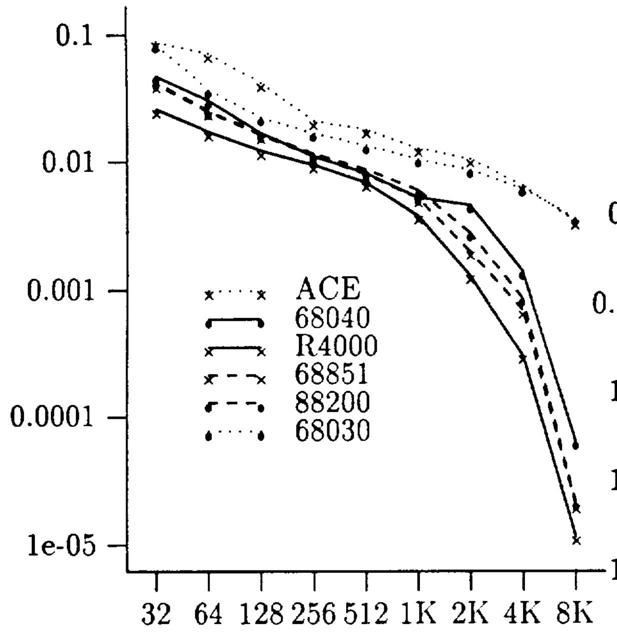

Figure A.20: mp3d TLB miss rate vs. page size (log scales)

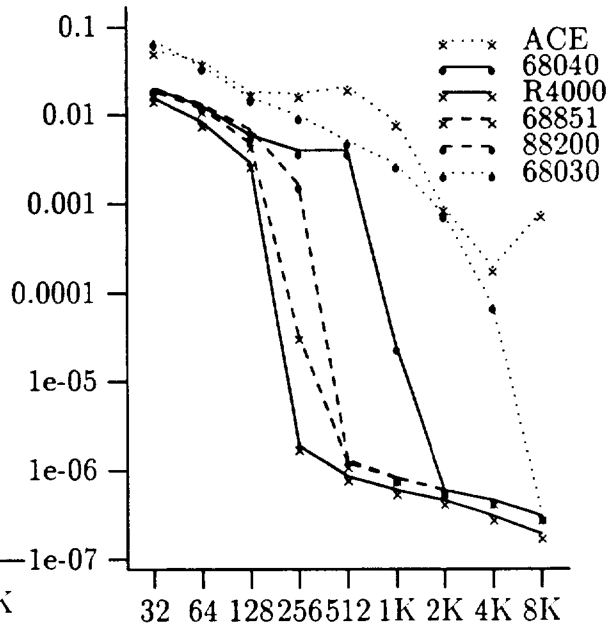

Figure A.22: e-hyd TLB miss rate vs. page size (log scales)

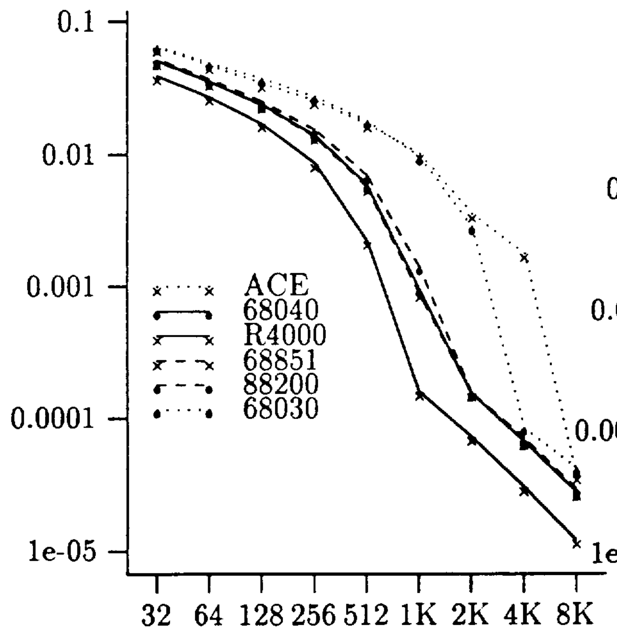

Figure A.21: bsort TLB miss rate vs. page size (log scales)

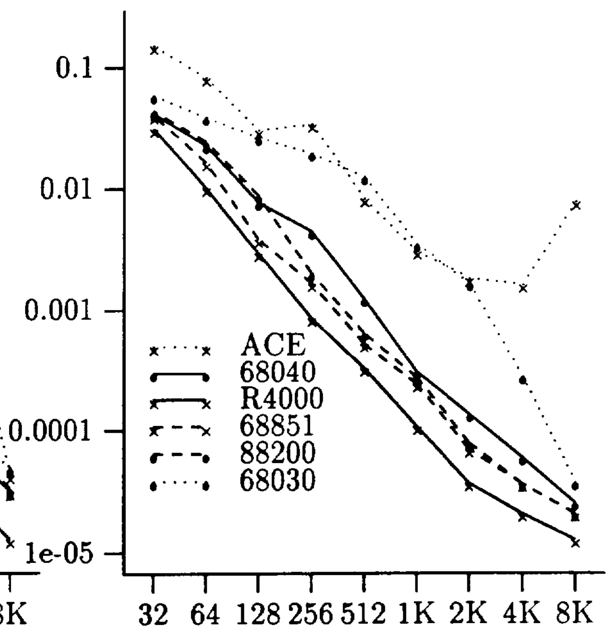

Figure A.23: p-gauss TLB miss rate vs. page size (log scales)

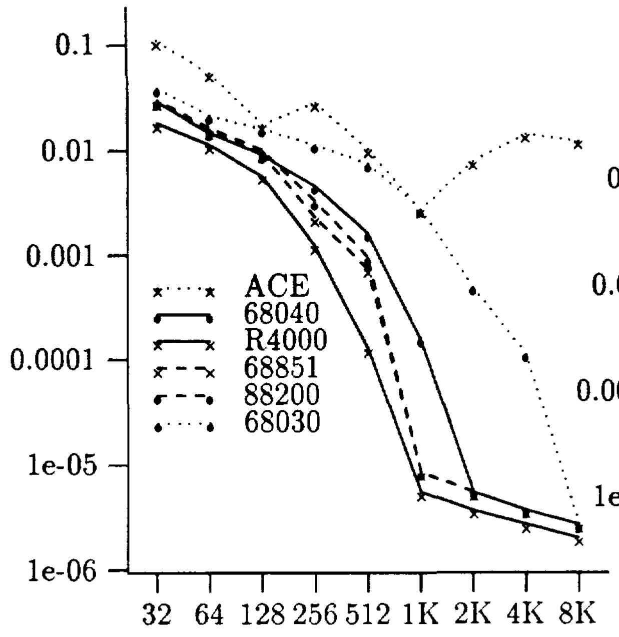

Figure A.24: **p-life** TLB miss rate vs. page size (log scales)

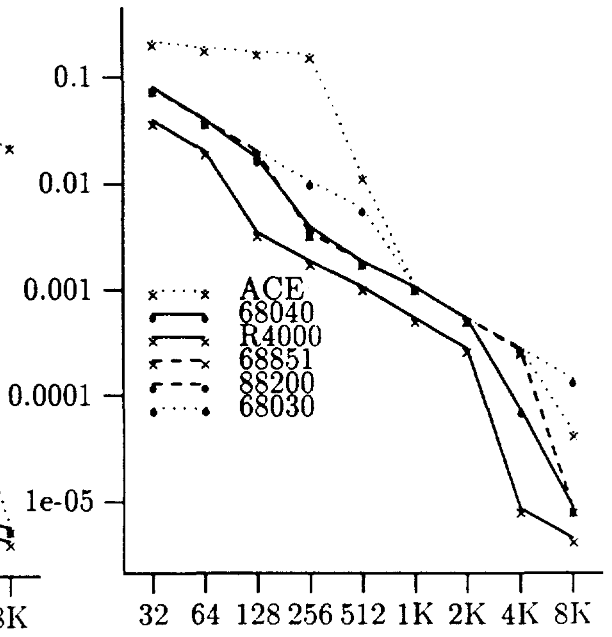

Figure A.26: **sorbyr** TLB miss rate vs. page size (log scales)

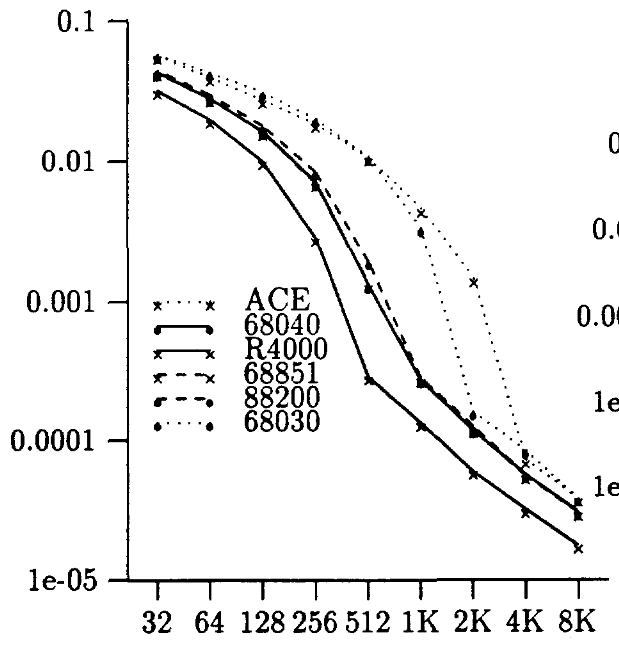

Figure A.25: **kmerge** TLB miss rate vs. page size (log scales)

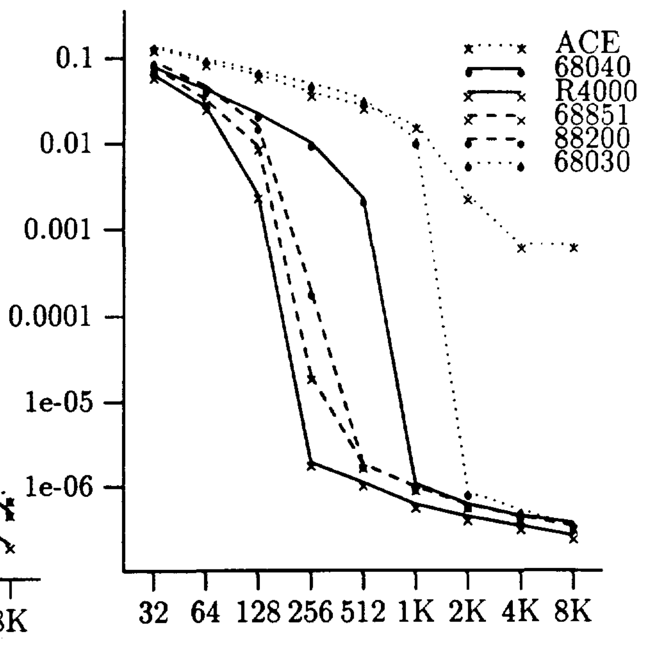

Figure A.27: **chip** TLB miss rate vs. page size (log scales)

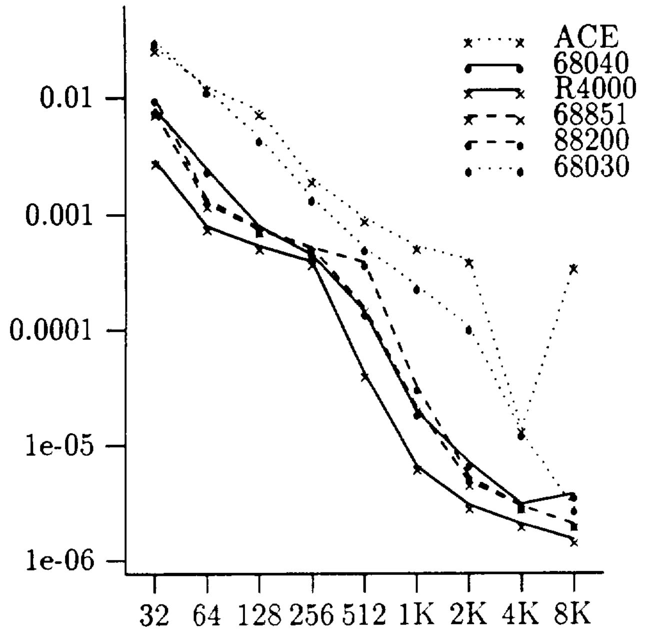

Figure A.28: **water** TLB miss rate vs. page size (log scales)

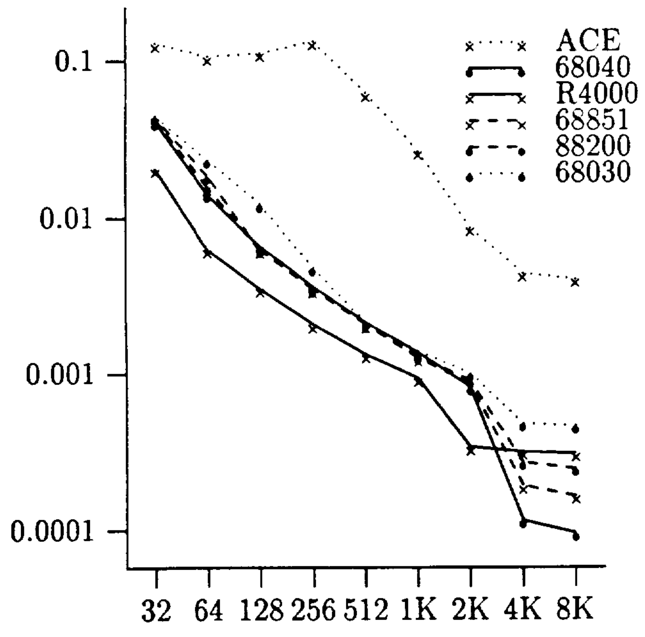

Figure A.29: **gauss** TLB miss rate vs. page size (log scales)