Dynamic Computation Migration in Distributed Shared Memory Systems

by

Wilson Cheng-Yi Hsieh

S.B., Massachusetts Institute of Technology (1988)
S.M., Massachusetts Institute of Technology (1988)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1995

Author . . . .
Department of Electrical Engineering and Computer Science
September 5, 1995

Certified by . . . .
M. Frans Kaashoek
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
William E. Weihl
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Dynamic Computation Migration in Distributed Shared Memory Systems

by
Wilson Cheng-Yi Hsieh

Submitted to the Department of Electrical Engineering and Computer Science
on September 5, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Computation migration is a new mechanism for accessing remote data in a parallel system. It is the partial migration of an active thread to data that it accesses, in contrast to data migration, which moves data to the computation that accesses it (often by making a copy). Computation migration improves performance relative to data migration, because data is not moved; as a result, it avoids sending protocol messages to keep replicated data consistent.

Performance measurements of two distributed shared memory systems demonstrate that computation migration is a useful mechanism that complements data migration. Computation migration performs well when data migration does not perform well; in particular, computation migration should be used instead of data migration when writes either are expensive or dominate reads.

The initial implementation of computation migration in the Prelude distributed shared memory system demonstrated that computation migration is useful for write-shared data, and that combining data migration and computation migration is useful for improving performance. The implementation of *dynamic computation migration* was done in the MCRL system. The decision to use data migration or computation migration in MCRL is deferred until runtime, when better knowledge of read/write ratios is available.

Experiments demonstrate that computation migration should be used for writes in two application kernels. For a concurrent distributed B-tree on Alewife, using computation migration for writes improves performance by 44%, relative to pure data migration, when the operation mix consists only of inserts. In addition, experiments demonstrate that dynamically choosing between data and computation migration for reads can improve performance. For a B-tree on Alewife with 80% lookups and 20% inserts, using a dynamic choice for reads improves performance 23% relative to pure data migration; always using data migration for reads only improves performance by 5%.

Keywords: computation migration, data migration, distributed shared memory, parallel programming

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor of Computer Science and Engineering

Thesis Supervisor: William E. Weihl
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

My advisors, Professors Frans Kaashoek and Bill Weihl, have been wonderful to work with, both professionally and personally. I certainly would not have finished if it were not for their guidance and knowledge, and I've learned a lot about doing research from them. They let me explore many different research areas, which made graduate school a great learning experience.

The rest of my committee has also been wonderful: John Guttag has always given me sound advice, and David Kranz has always been available to answer my questions. My committee provided incredibly useful feedback on my dissertation, which as a result is orders of magnitude better. Any mistakes, misstatements, or other problems with this dissertation remain mine, of course.

I thank everyone who acted as a reference for me during my job search: Bill Weihl, Frans Kaashoek, Fran Allen, Ron Cytron, and John Guttag. In particular, I'd like to thank them for writing me reference letters even after I had virtually decided to go to UW.

Ken Mackenzie, Don Yeung, and especially John Kubiatowicz (without whom there would be no Alewife machine) were extremely helpful in answering my numerous questions about the Alewife system. David Chaiken and Beng-Hong Lim, were willing to answer Alewife questions even after they graduated, and Kirk Johnson answered all of my questions about CRL. I also borrowed (with permission) some of Kirk's text and numbers about CRL. Finally, Scott Blomquist was great in keeping the CM-5 up and running.

Everyone in the PDOS research group made the 5th floor a fun place to work. My officemates Kevin Lew and Max Poletto put up with me this past year; I forgive Max for eating my plant, and for not listening to my advice to leave MIT. Debby Wallach answered all of my questions about using the CM-5, and was generous with her time in helping me on the CM-5; she was particularly good at shredding bad drafts of these acknowledgments. Anthony Joseph has always been more than helpful with questions of any sort, especially when they were not thesis-related. I wish Anthony and Debby the best of luck in getting out of here quickly. Dawson Engler got me interested in dynamic code generation: 'C would not have been designed were it not for his interest in the topic. Ed Kohler was a great source of typesetting and design knowledge, and he improved the look of my dissertation immensely.

Quinton Zondervan played chess with me when I needed breaks, and Parry Husbands answered all of my random theory questions. Bob Gruber, Carl Waldspurger, and Sanjay Ghemawat were all here with me at the beginning of this millennium; I wish them the best of luck in their respective careers, and I hope to see them often! Paul Wang had the common sense to know what he really wanted, and I hope he becomes a rich and famous doctor (medical, that is).

The Apple Hill Chamber Music Players have given me a wonderful atmosphere in which to play music during the last few summers. It was a refreshing change of pace to be there, and they helped to keep me sane. The music section at MIT is also very special, and I'll miss it deeply; it was a wonderful part of both my undergraduate and graduate careers. I thank everyone in the section for providing many great opportunities for making music.

I will miss my friends and fellow chamber musicians of the last few years: Eran Egozy, Don Yeung, Elaine Chew, Ronni Schwartz, and Julia Ogrydziak. When we play together again I'll try to be less insistent on playing things my way. Not. I have a great admiration for Jamie McLaren and Marc Ryser, who had the guts to leave MIT and become musicians.

Marcus Thompson, now Robert R. Taylor Professor of Music, has been a great teacher and friend for eleven years; I wish I could study viola with him for another eleven years. Being able to learn from him (about life as well as the viola) was a constant source of inspiration, and helped keep me sane through graduate school.

David Krakauer put up with me as a roommate for six years, even after I turned him down as a roommate in college. I owe him a great debt for letting me use his computer — especially when it was in his room and he was sleeping! Of course, Krak did his best to help me avoid my thesis: he always found computer games for me to play when I had work to do.

I give my deepest love to Helen Hsu, who has made my last year of graduate school far happier than it otherwise would have been. She has been my best friend as well as my biggest cheerleader, and I don't know if I would have finished this dissertation without her to cheer me up.

Last, but of course most importantly, I owe everything to my parents. They have my love and my gratitude for providing me with a wonderful upbringing and education, as well as their unfailing love and support.

Chapter 5 is a revision of text originally published in "Computation Migration: Enhancing Locality in Distributed-Memory Parallel Systems," by Wilson C. Hsieh, Paul Wang, and William E. Weihl, which appears in the *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

I was generously supported by a National Science Foundation Graduate Fellowship for the first three years of my graduate career. This research was also supported in part by many other sources: ARPA under Grants N00014-94-1-0985 and N00014-89-J-1988, an NSF National Young Investigator Award, Project Scout under ARPA contract MDA972-92-J-1032, NSF under Grant CCR-8716884, an equipment grant from DEC, and grants from AT&T and IBM.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation investigates *computation migration*, a new mechanism for accessing remote data in a parallel system. Computation migration moves computation to data that it accesses, in contrast to data migration, which moves data to computation (often by making a copy of the data). This dissertation investigates the design issues involved in using computation migration, and describes two implementations of computation migration in distributed shared memory systems. Performance measurements of these systems demonstrate that computation migration is a useful mechanism that complements data migration. In particular, computation migration is particularly appropriate for write operations in applications that use graph traversal.

Computation migration is the partial migration of active threads. Thread state is moved, but not code; the assumption is that the same program is loaded on every processor. Under computation migration, a currently executing thread has some of its state migrated to remote data that it accesses. Others have explored the migration of entire threads for load balancing, but it is not effective for accessing remote data because the granularity of migration is fixed (and typically too large). Computation migration generalizes thread migration by moving only *part* of a thread. In particular, computation migration moves the part of a thread that corresponds to one or more activation frames at the top of the stack. Reducing the amount of state that must be moved for a remote access makes computation migration a cheap mechanism for accessing remote data.

Computation migration performs well when data migration with replication does not. Conversely, computation migration performs poorly when data migration with replication performs well. Data migration with replication improves the performance of read operations (operations that do not modify data), because it allows them to execute in parallel. However, data migration with replication hurts the performance of write operations (operations that modify data), because data must be kept coherent. Computation migration outperforms data migration when there are many writes, because it sends fewer messages: when data is not moved, coherence traffic is eliminated.

Reducing the number of messages sent improves performance for two reasons. First, it reduces the demand for network bandwidth, which reduces the occurrence of network hot spots. More importantly, it removes the various costs of message handling, on both the sending and receiving sides. These costs include marshaling and unmarshaling values, allocating buffers, and dispatching to message handlers.

As part of the coherence traffic that it eliminates, computation migration also avoids moving data. Data migration must move data from processor to processor; in the case of large data (where "large" is relative to the bandwidth of the communication network), simply moving the data from one processor to another can be a major cost.

The remainder of this chapter overviews the research described in this dissertation. Section 1.1 explains the context of this work, distributed shared memory systems. Section 1.2 overviews the systems that I have built to evaluate computation migration. Section 1.3 summarizes my contributions and conclusions. Finally, Section 1.4 outlines the structure of this dissertation.

1.1 Context

This dissertation evaluates computation migration in the context of distributed shared memory (DSM) systems. A DSM system is a parallel system with two defining characteristics: memory is physically distributed along with the processors, but the system provides a global shared name space that can be used on any processor. Distributed shared memory systems are important for three reasons. First, they provide for the exploitation of locality; the distribution of memory allows for the distinction between local and remote data. Second, they provide a convenient programming model; the presence of a shared name space simplifies programming. Finally, many of them provide a portable interface for parallel programming.

In order to support local caching of data, most distributed shared memory systems typically use data migration with replication when non-local data is accessed. Data migration means that non-local data is brought to the thread that references it (whether through a pointer indirection or a procedure call); replication means that a copy of the data is brought. Not all systems that provide data migration use replication, but most do in order to improve performance.

Replication leads to the problem of *coherence*: multiple copies of data must be kept "in sync" so that the address space behaves like a single memory. Although distributed shared memory systems provide a simple (and possibly portable) model of programming, they often cannot achieve the performance of hand-coded message-passing programs. DSM systems typically use only one protocol for maintaining the coherence of data, whereas the access patterns for data vary widely. The Munin project [9] is unique in that it provides various different coherence protocols that can be used for different types of data. However, as in other DSM systems, Munin restricts its attention to the migration of data, and does not allow for the migration of computation.

This dissertation introduces a new mechanism that should be provided in distributed shared memory systems, computation migration, which complements data migration with replication. The choice of mechanism for any particular remote access depends on an application's characteristics and on the architecture on which the application is being executed. Programmers (or preferably compilers) should be able to choose the option that is best for a specific application on a specific architecture.

1.2 Overview

In initial work we designed and implemented a prototype of static computation migration. "Static computation migration" means that the programmer statically indicates in the program where computation migration occurs. This prototype was implemented in the Prelude distributed shared memory system [100]. In Prelude we used compiler transformations to implement computation migration based on simple annotations.

This dissertation overviews our Prelude implementation and some of our performance results on a concurrent, distributed B-tree, which lead to two conclusions. First, a software implementation of static computation migration can performs nearly as well on a B-tree as a hardware implementation of data migration with replication. Second, a combination of computation migration and data migration could potentially outperform data migration.

In more recent work, I have designed and implemented a protocol for making a dynamic choice between computation migration and data migration. "Dynamic computation migration" means that the programmer indicates where computation migration may occur; the decision to use data migration or computation migration is deferred until runtime. This dissertation discusses the issues involved in designing such a protocol, as well two heuristics that make good choices between computation and data migration.

The protocol for dynamic computation migration is implemented in the MCRL distributed shared memory system. MCRL is a multithreaded object-based DSM library that provides support for variable-sized, programmer-defined regions of memory. Using a system that supports variable-sized regions avoids the false sharing of data, which can occur in systems that maintain coherence in fixed-sized units. Several DSM systems have gone to great lengths to avoid false sharing in their page-based coherence schemes [19, 59], so it is important to avoid letting false sharing obscure other performance effects.

This dissertation describes the implementation of MCRL, and the results of measuring its performance.

1.3 Contributions

This section summarizes my contributions and conclusions:

- I present computation migration, a new mechanism for accessing remote data in parallel systems. Computation migration complements data migration, in that it performs well when data migration does not.

- I describe how a simple annotation can be provided to express computation migration of single activation records. Such an annotation can be used to indicate a static decision to use computation migration, or can be used as a hint to indicate that a dynamic decision between data migration and computation migration should be made.

- I have built a multithreaded distributed shared memory system called MCRL that incorporates dynamic computation migration. MCRL is a multithreaded extension of the CRL system [51];

it also supports a dynamic choice between computation migration and data migration. MCRL is an all-software distributed shared memory system: it does not depend on any special hardware support to provide shared memory. MCRL runs on the MIT Alewife machine and Thinking Machines' CM-5.

- I describe and evaluate two simple heuristics for choosing between computation migration and data migration, STATIC and REPEAT. My experiments with a microbenchmark demonstrate that simple heuristics can make good choices between data migration and computation migration for reads.

- My experiments demonstrate that computation migration should be used for write operations in the two application kernels measured. For a concurrent distributed B-tree on Alewife, using computation migration for writes improves performance by 44%, relative to pure data migration, when the operation mix consists only of inserts. In addition, my experiments demonstrate that dynamically choosing between data and computation migration for reads can improve performance. For a B-tree on Alewife with 80% lookups and 20% reads, using a dynamic choice for reads improves performance 23% relative to pure data migration; always using data migration for reads only improves performance by 5%.

- A comparison of MCRL on Alewife and the CM-5 provides some insight into architectural issues that impact data migration and computation migration. First, DMA is an important mechanism for data migration. Second, restrictions on message length need to be chosen with care, as they can severely limit performance.

- Finally, this dissertation provides some additional insight into the benefits and restrictions of using optimistic active messages. Optimistic active messages are a communication mechanism that generalizes active messages [95]; we have demonstrated its usefulness in other work [49, 97].

1.4 Organization

Chapter 2 overviews several remote access mechanisms: remote procedure call, data migration, and thread migration. It then briefly describes computation migration, and how it compares to the other mechanisms.

Chapter 3 discusses computation migration in more detail. It analyzes the design issues regarding computation migration, discusses why a dynamic choice between computation migration and data migration is useful, and describes two heuristics for making such a choice.

Chapter 4 describes the experimental environment for this dissertation. It overviews the platforms on which I evaluated computation migration: the MIT Alewife machine, Thinking Machines' CM-5, and a simulated version of Alewife on the Proteus simulator.

Chapter 5 describes our initial work on static computation migration. It describes the implementation of static computation migration in the Prelude system, summarizes some of the performance results, and explains why dynamic computation migration can be useful.

Chapter 6 describes the implementation of MCRL and the interface for dynamic computation migration that MCRL provides. It describes the differences between CRL and MCRL, as well as various implementation issues. It concludes by describing some of the implementation details on Alewife and the CM-5.

Chapter 7 describes and analyzes the performance results of using dynamic computation migration in MCRL. It measures the raw performance of MCRL, as well as the behavior of the two heuristics on a microbenchmark and two application kernels.

Chapter 8 discusses related work. Chapter 9 summarizes the results and conclusions of this dissertation and suggests directions for future work.

The appendices provide supplementary information. Appendix A gives some technical details on the internals of the CRL and MCRL systems. Appendix B explains the effect that LIFO scheduling of threads can have on measurement. Appendix C contains the linear regression formulas used to calculate my performance results.

Chapter 2

Remote Access Mechanisms

This chapter describes various forms of remote access: remote procedure call, data migration, thread migration, and computation migration. It then briefly compares computation migration to data migration and remote procedure call.

2.1 Remote Procedure Call

The most common mechanism for implementing remote access in a message-passing system is remote procedure call (RPC for short), which has been used in both distributed and parallel systems [12]. RPC, which is also referred to as "function shipping," is typically the choice of interface for client-server systems. Part of the reason is that an RPC interface provides protection through encapsulation (analogous to the protection that object-oriented languages provide).

RPC's resemble local procedure calls; the underlying message-passing is hidden by compiler-generated stubs. When an RPC occurs, a local call is made to a client stub, which marshals the arguments and sends a message to the processor where the data resides. At the remote processor, a server stub unmarshals the arguments and calls the procedure. When the remote procedure returns to the server stub, the server stub marshals any result values into a message and returns the message to the client stub; the client stub returns the results to the RPC caller.

Remote procedure calls require two messages, one for the call and one for the reply. Using RPC for a series of accesses to remote data can result in no locality of access; each successive access is remote. Figure 2-1 illustrates the message-passing patterns of RPC, where one thread on processor P_0 makes n consecutive accesses to each of m data items on processors 1 through m, respectively. A total of $2nm$ messages are required, ignoring coherence traffic.

One advantage of RPC is that it does not add excessive load on a server. That is, when a remote call finishes and returns, the thread that initiated the call does not consume any more resources on the server. Another advantage of RPC is that it performs well in the case of write-shared data (shared data that is frequently written). Since the semantics of writes requires that they be serialized, there is no parallelism to be gained by replicating data. As a result, when the pattern of writes is unpredictable, it is faster for writers to use RPC instead of moving data from writer to writer.

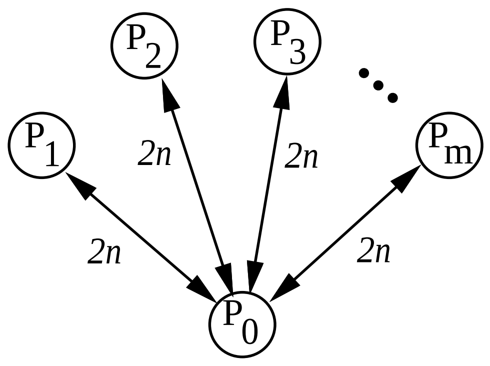

Figure 2-1. Message pattern under remote procedure call. A thread on processor P_0 makes n consecutive accesses to each of m data items on processors 1 through m, respectively. Coherence traffic is not shown.

2.2 Data Migration

Data migration, also known as "data shipping," means that remote data is moved to the processor where a request is made. Data migration can be combined with replication, where a copy of the data is sent to the requesting processor; this introduces the problem of *coherence*. In the remainder of this dissertation I use "data migration" to include the use of replication, and explicitly state when replication is not involved. Data migration can take the form of hardware caching in shared-memory multiprocessors such as Alewife [1] and DASH [66].

Data migration can also be implemented in software. For example, the Munin system [9] allows the programmer to choose among several different coherence protocols for shared data; some of the protocols use replication, and some do not. The Emerald system [55] is one of the few systems that uses data migration without replication. The use of replication is important for performance, however, particularly for read-shared objects, as it allows locally cached copies to be accessed without network overhead. Figure 2-2 illustrates the message-passing patterns of data migration, where one thread on processor P_0 makes n consecutive accesses to each of m data items on processors 1 through m, respectively. A total of $2m$ messages are required, ignoring other coherence traffic.

Data migration can perform poorly under several circumstances. First, large data is expensive to migrate. Second, data that is written with any frequency is expensive to keep coherent. Write-shared data appears to occur moderately frequently, and can result in data that migrates from one cache to another with relatively few copies existing at any one time [40]. Some researchers have tried to optimize their cache-coherence algorithms for so-called "migratory data" [30, 89] to reduce the communication required to maintain consistency. Even so, RPC and computation migration require fewer messages than data migration for write-shared data, and can result in better overall performance.

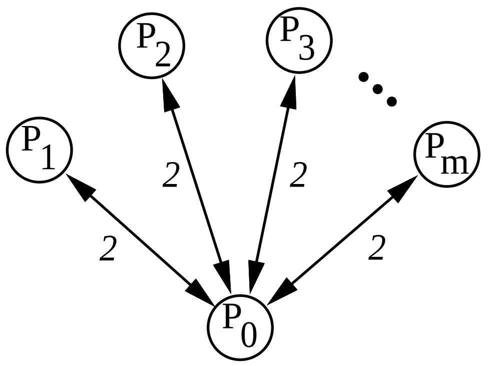

Figure 2-2. Message pattern under data migration. A thread on processor P_0 makes n consecutive accesses to each of m data items on processors 1 through m, respectively. Coherence traffic is not shown.

One important advantage of data migration combined with replication is that it can improve the locality of repeated accesses. After the first access by a thread to some data, successive accesses will be local, assuming that a thread on another processor does not invalidate the first thread's cached copy. In addition, data migration and replication increase the possible concurrency for accessing read-shared data (shared data that is rarely written); multiple copies of data can be read in parallel.

2.3 Thread Migration

Thread migration is a mechanism that has primarily been examined as a load balancing technique [24]. "Thread" is used in the sense of lightweight processes [28, 75], which contrasts with "run-to-completion" threads in systems such as TAM [85], Cilk [13], Filaments [34, 36], or Multipol [22]. Run-to-completion threads are even more lightweight than lightweight processes; however, they either cannot block or may only block in a limited set of circumstances.

Thread migration is analogous to process migration [55, 79, 87], but is more lightweight, because processes have much more state than threads. Thread and process migration can occur in two forms: active migration, where currently executing threads or processes are migrated, and inactive migration, where non-executing threads or processes are migrated.

Eager *et al.* [32] used analysis and simulation to investigate the performance benefits of migrating active processes to improve load balance. They found that active process migration has limited performance benefits over inactive process migration. Their results apply to thread migration as well, since they show that ignoring the cost of migration does not change the qualitative nature of their results.

Although active thread migration is of limited use in balancing load, and not widely used, many systems use inactive thread migration to balance load. To name just a few, the Topaz operating

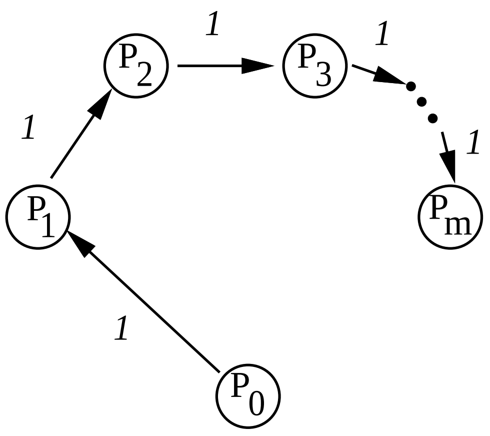

Figure 2-3. Message pattern under computation migration. A thread on processor P_0 makes n consecutive accesses to each of m data items on processors 1 through m, respectively. Coherence traffic is not shown.

system [72] for the Firefly multiprocessor workstation [92] migrates inactive threads. Markatos [70] explored a scheduling policy that favors locality over load balance: threads are initially scheduled based on expected accesses, and idle threads are migrated to balance load. Anderson *et al.* [3] studied, among other things, the performance implications of using local queues for waiting threads; threads in these queues could be migrated to balance load.

None of these systems explored the possibility of using active thread migration to improve the locality of a remote access. The major problem with such an approach is that the grain of migration is too coarse: migrating an entire thread can be expensive. As a result, migrating an entire thread for a remote access may be overkill for two reasons. First, much of a thread's state may be unrelated to the access. Second, migrating an entire thread could cause load imbalance.

2.4 Computation Migration

Computation migration is a generalization of active thread migration, in which a portion of a thread migrates upon a remote access. Migrating only part of a thread reduces the granularity of migration, which alleviates the two problems of thread migration. First, we can migrate only the state relevant to a remote access. Second, we can avoid load imbalance by moving only small pieces of computation.

Figure 2-3 illustrates the message-passing patterns of computation migration, where one thread on processor P_0 makes n consecutive accesses to each of m data items on processors 1 through m, respectively. A total of $m-1$ messages is required (plus one additional message if the call must return to P_0), ignoring coherence traffic. Computation migration requires only "one-way" messages, because unnecessary return messages are not sent. This optimization is a form of tail forwarding [47], which is a generalization of tail recursion to parallel systems.

Computation migration also gives some of the benefits of thread migration and RPC. If we move part of a thread's stack to the remote data, we can receive the benefit of increased locality of accesses; for example, if the executing thread makes a series of accesses to the same data, there is a great deal to be gained by moving those accesses to the data. At the same time, we gain the benefit of RPC; we can vary the granularity of the computation to be performed at the data. Keeping the computation small allows us to avoid the problem of overloading the resources at a single processor, since we move only the amount of state necessary to improve locality.

The cost of computation migration depends on the amount of computation state that must be moved. If the amount of state is large (such as a substantial portion of a thread) computation migration will be fairly expensive. In addition, when the amount of data that is accessed is small or rarely written, data migration with replication (especially when done in hardware) should outperform computation migration.

Computation migration is the dual of data migration. Instead of moving data to computation that accesses the data, the system moves the computation to the data. If a computation accesses different data, not all of which are on the same processor, some data migration may need to occur. In some cases it may be better to migrate the computation as well as some of the data; the pure data migration solution of always migrating all of the data to the computation may not always be the best.

2.5 Comparison

When comparing computation migration and data migration, the most obvious performance benefit is that computation migration does not move data. Although the resulting increase in performance can be great for very large data, it is not the primary reason that computation migration can outperform data migration. Figures 2-1 through 2-3 illustrate how computation migration can save messages when traversing a data structure, but that savings is also not the major benefit of computation migration. The major benefit of computation migration is that coherence traffic is eliminated.

The reason that computation migration reduces coherence traffic is because a migrated computation leaves the data in place for the next migrated computation. That is, the performance gain results from keeping the data at one processor. As a result, successive migrated computations need not perform any coherence activity. Although specialized coherence protocols can sometimes be used to reduce the cost of coherence (as in Munin [9]), they do not help when data is accessed unpredictably.

Although computation migration does not incur the cost of maintaining coherence, it has the disadvantage that it does not allow for replication. Replication allows reads from multiple processors to occur locally and in parallel, which dramatically improves their throughput. As a result, computation migration is more effective for write-shared data, whereas data migration with replication is more effective for read-shared data.

Finally, remote procedure call can be viewed as a special case of computation migration in which no thread state is migrated. As a result, using RPC can save coherence messages, just as computation migration does. The use of computation migration, however, can make successive accesses by a thread local; the use of remote procedure call means that successive remote procedure calls are non-local. (Section 8.3 describes research that investigates the batching of RPC's so as to reduce communication costs.) Similarly, computation migration saves messages by "short-circuiting" return messages; when a computation migrates and then executes a procedure, the procedure can return locally to its caller.

Chapter 3

Computation Migration

The use of data migration treats a parallel machine as a two-level memory hierarchy: from the point of view of any processor, there is local memory and remote memory. However, a distributed-memory parallel machine is not just a memory hierarchy: computation can move from one memory's processor to another memory's processor. Remote procedure call allows computation to be initiated at a remote processor, but does not allow an existing computation to migrate. This chapter discusses high-level issues regarding computation migration: how much computation should be migrated, when computation should be migrated, how computation migration can be expressed, and how it can be implemented.

Section 3.1 discusses how computation can be migrated. Section 3.2 discusses how simple program annotations can be used to indicate the use computation migration. Finally, section 3.3 discusses how the decision to use computation migration can be made.

3.1 Mechanics

Computation migration is partial thread migration. The first issue that must be resolved is the granularity at which computation is migrated. The simplest solution is to migrate a single activation record. Not only is it the unit in which a stack is usually allocated, but it is also easy to express the migration of an activation record, as described in Section 3.2.

It would be possible to migrate either multiple activation records, or just a portion of an activation record. The activation record/procedure unit is a unit of measure determined by the abstractions in a program, and may not always be the correct unit of migration. For example, it may be useful to migrate just the execution of a single loop in a procedure, instead of migrating the entire procedure.

Issues relating to the migration of partial or multiple records are discussed as they arise. The implementations of computation migration described in this dissertation only migrate single activation records. However, there is no fundamental reason why this research could not be generalized to migrate partial or multiple records.

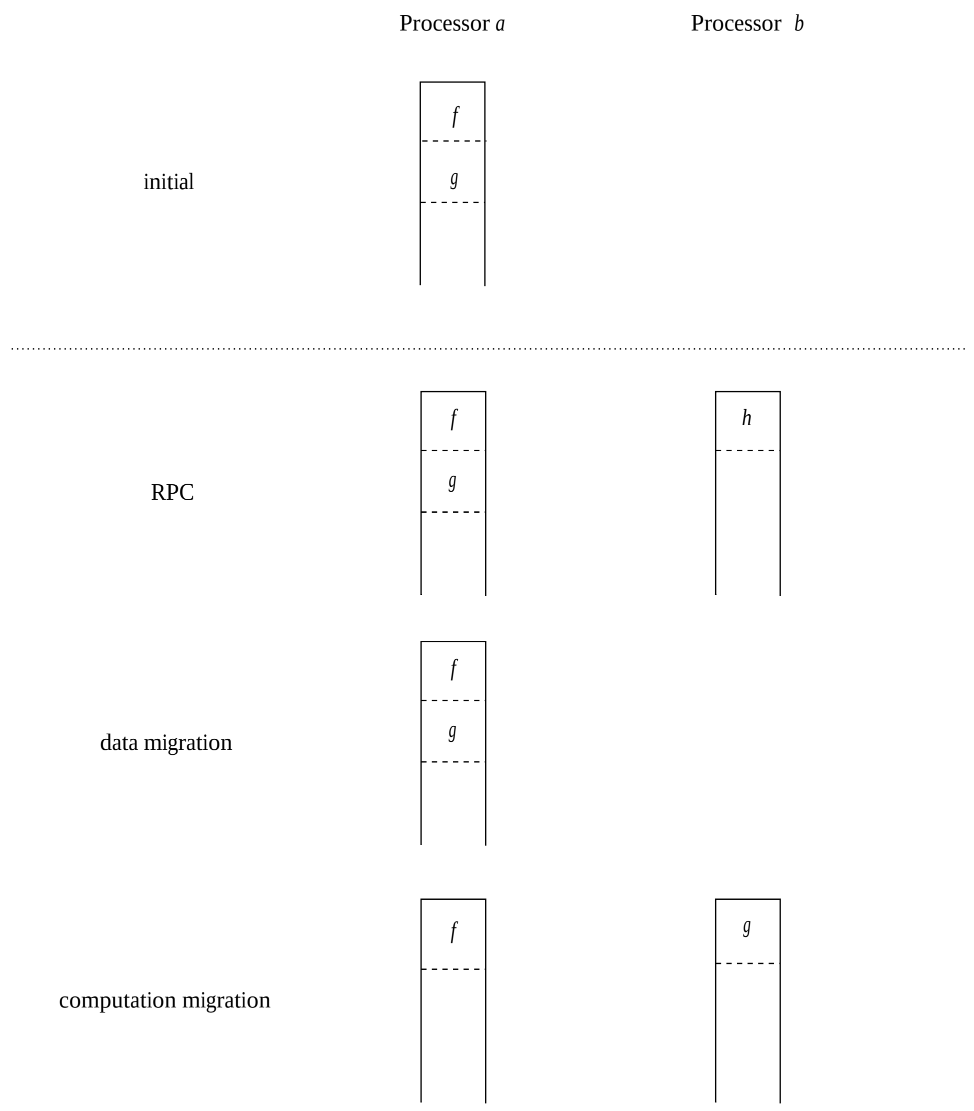

Figure 3-1. Stack patterns under different access mechanisms. Processor *a* is executing a thread, where *f* has called *g*. *g* is about to access remote data on processor *b*. Stacks grow downward in this figure.

Figure 3-1 illustrates the movement of activation records under RPC, data migration, and computation migration. In the figure, "initial" is the starting state. Processor *a* is currently executing a thread. In this thread, the procedure *f* has called the procedure *g*. *g* is about to access remote data on processor *b*. Under RPC, *g* calls a remote procedure *h* that runs in a new thread on processor *b*; this procedure returns its value to *g*. Under data migration, the data is moved or copied to processor *a*, and execution continues on processor *a*. Under computation migration, the activation record for *g* is moved to processor *b*, where the access to the data occurs.

In RPC implementations, compiler-generated "stubs" marshal arguments and return results. The stubs hide the details of communication from the actual code for *g* and *h*, so that the compiler can generate code for them normally. In data migration implementations, either the compiler, library, operating system, or hardware is responsible for moving data.

In computation migration implementations, some mechanism must be provided to move activation records. To move activation records safely, we must restrict the use of computation migration so that a processor does not export non-global memory addresses. This restriction only exists for implementations of computation migration that are not run on shared-memory systems.

The remainder of this section describes two implementations of computation migration. These two implementations have complementary strengths and weaknesses, and could be used together in one implementation. The first implementation described relies on the runtime system to implement migration, and has been implemented in the Olden system [81]. The second relies on the compiler to generate code to handle migration, and was implemented in Prelude [50]. MCRL provides an interface for a compiler-based implementation, but does not provide compiler support for using it.

3.1.1 Runtime System

The most direct implementation of computation migration is to have the runtime system move an activation record. In other words, the runtime system must contain enough functionality to marshal an activation record, send it over the network, and unmarshal the activation record onto a stack. After setting a return stub, the runtime system can then return control directly to the migrated code. When the migrated code returns, the stub is responsible for returning control back to the original processor. The Olden project [17] takes this approach in implementing computation migration.

This implementation is flexible, in that multiple activation records can be migrated in a straightforward manner. However, it suffers from several complications. First, a simple implementation will move some unnecessary data; not all of the data in an activation record will be live at the time of migration. In order to avoid moving dead data, the interface for migrating an activation record must be fairly complex. The compiler must generate a template for each migration point that indicates which entries in an activation record are live. Second, even if moving dead data is not an issue, a similar issue arises if a partial frame needs to be moved. In order to move a partial activation record, the compiler would again need to tell the runtime system what slots in a record to move.

3.1.2 Compiler

Another approach to implementing computation migration is to have the compiler (or user) explicitly generate migration procedures. Such an approach can be viewed as having the compiler generate specialized migration code for each procedure. The compiler must generate a new procedure for each migration point. This procedure represents a “continuation” of the original procedure, in that it contains the code to be executed from the migration point to the end of the procedure. This continuation procedure is the code that is executed after a migration occurs. The data that needs to be moved to execute the continuation procedure is the set of live variables at the migration point.

This approach can produce better migration code, in that only the portion of the stack frame that needs to be migrated is actually transmitted. In addition, this approach is easily generalized to move a partial activation record. However, it suffers from two drawbacks. First, it leads to code expansion. By producing specialized code for each migration point, the compiler winds up duplicating a large amount of code. Second, generalizing this approach to migrate multiple frames requires some extra mechanism. For example, one way to extend this approach to the migration of multiple frames would be to pass the proper continuation procedure at every procedure call. Such a solution would risk making the common case (any procedure call) more expensive to support computation migration.

3.2 Program Annotation

There are several important reasons to use annotations for computation migration. First, it is simple to experiment with computation migration. Changing where migration occurs simply involves moving the annotation, and the programmer can easily switch between using computation migration, RPC, and data migration. Second, the annotation affects only the performance of a program, not its semantics. In other words, the annotation only affects where computation executes.

Some languages permit a programmer to code a structure explicitly that mimics the effect of computation migration. For example, Actor-based languages [69] encourage a “continuation-passing” style for programs, in which each actor does a small amount of computation and then calls another actor to perform the rest of the computation. In effect, the computation migrates from actor to actor. By carefully designing the message-passing structure of the program, a programmer can optimize the communication cost.

For several reasons, computation migration should not be coded explicitly into the structure of a program. The most important reason is that such programs are difficult to tune and port. Performance decisions in such a program are closely intertwined with the description of the algorithm; changing the communication pattern may require substantial changes to the program, and may result in inadvertent errors being introduced into the computation. It is better to separate the description of the computation from the decisions of how it is to be mapped onto a particular machine, and to let the compiler handle the details of the message-passing for computation migration.

Finally, experience with both parallel and distributed systems indicates that programming in terms of explicit message-passing can lead to programs that are much more complex than ones written in terms of shared memory. An explicit message-passing style is also undesirable from

```
void traverse(node *n)
{
  read_lock(n);
  while (! is_leaf(n)) {
    next = next_node(n, k);
    read_unlock(n);
    n = next;
    read_lock(n); move  /* annotation */
  }
  read_unlock(n);
}
```

Figure 3-2. Sample annotation for computation migration. The annotation indicates that the procedure `traverse` should use computation migration to access `n` if it is remote.

a software engineering perspective, because it requires the modification of the interfaces of data objects to include additional procedures that represent the migrated computations.

Figure 3-2 illustrates how a simple annotation can be used to indicate computation migration in a C-like pseudocode. The annotation indicates that at the bottom of each iteration of the `while` loop, the computation should migrate to the next node if the node referenced by `n` is not local. The entire activation record moves; that is, the computation from the `move` annotation to an exit from `traverse` executes on the remote processor, unless it migrates again to yet another processor.

A simple annotation for migrating a partial frame is illustrated in Figure 3-3. Two annotations are used; one to indicate where a migration should begin, and one to indicate where a migration should return. In the figure, the annotations indicate that the code between `read_lock` and `write_lock`, inclusive, is the code to be migrated. If `n` is non-local when `read_lock` is called, that block of code should execute on the processor where `n` is located. When the `read_unlock` operation completes, control should return back to where `traverse2` was executing.

A simple annotation for migrating multiple activation records is difficult to design, and is left for future work. Moving a single or partial record is easier to specify, since the textual representation of an activation record is captured by a single procedure. In addition, the migration of multiple records should be flow-sensitive; that is, multiple records should be migrated for certain call chains, but not for others.

3.3 Migration Criteria

This section describes how the decision to use computation migration could be made. First, it could be made statically, based on programmer knowledge about object sizes and reference patterns. If such knowledge is insufficient, a dynamic choice is necessary.

```
void traverse2(node *n)
{
  read_lock(n);     begin move /* annotation */
  /*
    some operations on n
   */
  read_unlock(n);   end move  /* annotation */
}
```

Figure 3-3. Sample annotation for computation migration of a partial frame. The annotation indicates that the procedure `traverse2` should use computation migration to access `n` if it is remote when `read_lock` is called. The code should migrate back after `read_unlock`.

3.3.1 Static

The simplest strategy for providing computation migration is for the programmer to decide statically where computation migration should occur in a program. The decision can be indicated with a simple annotation, such as the one described in Section 3.2. The decision to use computation migration is based on the knowledge that computation migration outperforms data migration with replication under either of the following conditions:

- The object being accessed is large, so moving it rather than the computation is expensive.
- The object being accessed is written often, so maintaining coherence is expensive.

Although the first condition is easy to detect, the second may not be. More importantly, most objects are likely to be small, so the more important factor in determining whether to use computation migration is likely to be the second. As a result, except in the case of write-only or write-mostly data structures, it may be difficult for a programmer to know at compile-time that a remote access should always be performed using computation migration.

Another difficulty with statically determining when to use computation migration is that the same code may be used to access either a read-shared or a write-shared object. For example, in Figure 3-2, it is not possible to know whether the variable `n` will refer to a read-shared or a write-shared object. It would only be possible to know if the objects on which `traverse` is called are either all read-shared or all write-shared. This condition may hold for certain applications. For example, the counting network, which I describe in Section 7.3.1, is a completely write-shared data structure. However, for general data structures it is unlikely that the condition is true.

3.3.2 Dynamic

If the programmer cannot statically determine when computation migration should occur, then the logical choice is to choose dynamically when to use computation migration. Since the runtime system can monitor the reads and writes of an object, it can determine when computation migration would be beneficial. The programmer still uses an annotation, but the annotation becomes a hint. The runtime system is responsible for deciding whether the hint should be followed.

Unfortunately, past performance is not a guarantee of future performance. In other words, it is not in general possible to predict the future frequency of writes based on the past frequency of writes. However, in order to make a dynamic decision tractable, we must assume that the relative frequencies of reads and writes stay relatively constant for over short periods of time. Compile-time information can sometimes be used to predict future access patterns [5], but the information that can be gained using current techniques is limited.

We could use many metrics to determine the frequency of reads and writes. For example, any of the following might be useful:

- The average number of reads between writes. This measure would require some bookkeeping on every operation.
- The relative frequency of reads and writes. This measure requires a mechanism for counting time, which is difficult. It also requires some bookkeeping on every operation.
- The average number of read copies when a write begins. This measure is equivalent to the average number of invalidations that a write must perform.
- The average cache hit rate. Computing the hit rate requires bookkeeping on local operations, which is undesirable.
- The number of threads on each processor. Keeping count of the number of threads gives an indication of the load balance.

All of these metrics are fairly complex. They also require a mechanism for weighting more recent measurements more heavily than older measurements. Although the above metrics may lead to good decisions, the cost of computing them is relatively high. The following section describes two simple heuristics that make relatively good choices.

3.3.3 Dynamic Protocol

The first question that arises when designing a protocol for dynamically choosing between data and computation migration is which processor makes the choice. When a client accesses non-local data of which it has a cached copy, it need not perform any protocol processing. However, when it does not have a cached copy, it must contact the server. When the client contacts the server, either processor could decide between data or computation migration:

- The client can decide whether to use data or computation migration. There is no overhead to using either mechanism beyond the decision itself.

- The client can request that the server decide to use data or computation migration. There is some overhead involved for either mechanism, since the client must pass enough information to allow the server to make the decision.

My protocol uses a server-based choice because the server has the most information about the pattern of global accesses to data. Although a client can keep track of its own access patterns, it does not have any information about other clients' access patterns. Even if the server regularly broadcasts global information to clients (which could potentially be expensive), a client never has information that is as up-to-date as the server. In addition, since the client must contact the server anyway, it seems advantageous to make use of the server's extra knowledge of global access patterns.

The protocol does not add any extra messages for using data migration, but does add an extra message for computation migration. It may be possible to design a dynamic migration protocol that does not require this extra acknowledgment, but this design point was chosen for the sake of simplicity.

A client and server interact in the protocol in the following manner. When a client accesses a local object (either its home is local or it is cached), the local copy can simply be used without any communication. When a client accesses a remote object, it sends a *migration request* message to the server. (It must also send such a message if it tries to acquire a read-only copy of a region in write mode.) This message must contain enough information for either data migration or computation migration to occur. That is, the processor must be in a state such that it can receive either a copy of the data or a migration acknowledgment, which I describe below.

When the server receives a migration request message, it must first decide whether to send the data back or to execute some computation. If the server decides to use data migration, it sends the data back to the client. If the server decides to use computation migration, it must send back a "migration acknowledgment" to the client; it can then either create a thread to defer the computation, or execute it directly.

Figure 3-4 illustrates some possible message patterns using my dynamic protocol. When a processor receives a migration request message, it decides whether to send back data or notify its client that it has decided to execute computation. The dynamic computation migration protocol requires more bandwidth than static computation migration, since the migration acknowledgment must be sent. However, the cost of sending and receiving this extra message does not have to be on the critical path of the migration, as this acknowledgment can be sent in the background. In Figure 3-4 the same decision (to migrate data or computation) is made at every processor; it is also possible for different decisions to be made at each processor.

STATIC Heuristic

The STATIC heuristic always migrates computation for non-local writes and data for non-local reads. Ignoring considerations of load balance, it is intuitively a clear benefit to always migrate writes to data (assuming that the computation is not much larger than the data). Given the semantics of writes, they must always be serialized anyway. As a result, it makes sense to execute them all at one processor, which avoids the cost of moving data from one writer to another. The only case where this heuristic fails to perform well is when exactly one remote processor accesses the region (and

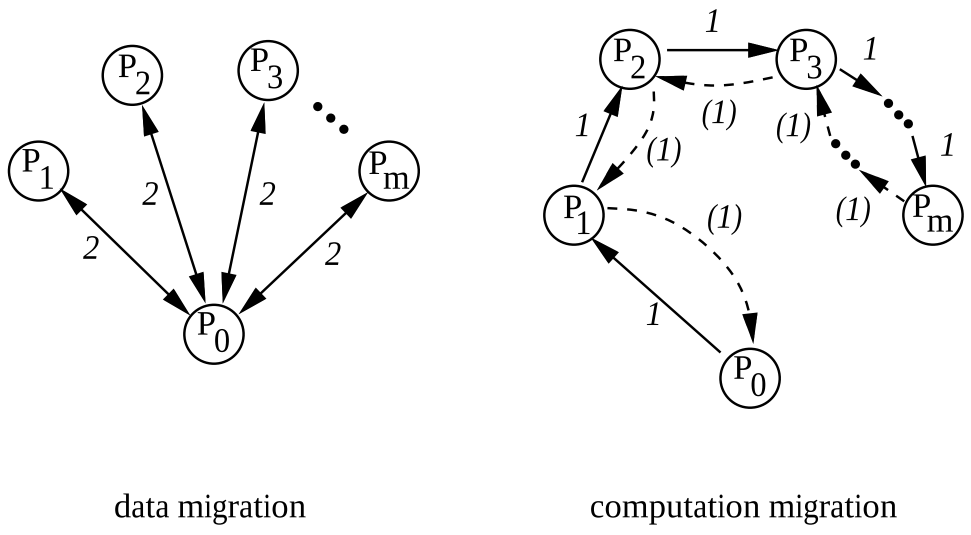

Figure 3-4. Message patterns under a dynamic protocol. Messages labeled in parentheses can be sent in the background. Coherence traffic is not shown.

the processor occasionally writes the region). Note that the STATIC heuristic is not truly a dynamic scheme; in most programs any use of a CRL function can statically be determined to be either a read or a write.

The decision to migrate computation or data in the case of reads is less clear. There is an inherent tension between computation migration and data migration combined with replication. If reads are computation-migrated to the home, performance is reduced when there are many more reads than writes: future reads do not hit in the cache. On the other hand, if reads migrate data back to themselves, performance is reduced when there is a substantial number of writes: future writes must invalidate more cached copies of data.

REPEAT Heuristic

The REPEAT heuristic always migrates computation for non-local writes, for the same reason that the STATIC heuristic does. It dynamically chooses data migration or computation migration for non-local reads, depending on the relative frequency of reads and writes. The heuristic uses computation migration for reads that follow writes, until any processor performs a second read before another write occurs. That is, if a processor manages to execute two reads before any writes occur, data migration should be used.

The logic underlying this heuristic is that data migration will be useful when replication will help, which is exactly when multiple reads occur at one processor. Given the assumption that the relative frequency of reads and writes remains relatively constant over several operations, we interpret the fact that two repeated reads occurred to mean that repeated reads are likely to occur in the near future.

Intuitively, this heuristic will perform better than the STATIC heuristic when there is a large proportion of writes. The STATIC heuristic migrates reads even when there are many writes. As a result, it forces writes to invalidate replicated copies unnecessarily. However, the REPEAT heuristic should do worse when the operation mix consists predominantly of reads, since it migrates some read operations following a write. As a result, the percentage of cache hits will decrease.

Chapter 4

Experimental Environment

This chapter describes the experimental environment for this dissertation. The evaluation of dynamic computation migration was done on two machines: the MIT Alewife machine and Thinking Machines' CM-5. The comparison of the two implementations demonstrates how communication performance and multithreading support affects DSM systems in general, and data migration and computation migration in particular.

- Although the CM-5's network has a greater total aggregate bandwidth into each processor, Alewife is able to use achieve a higher data transfer rate.
- The restriction on message length on the CM-5 dramatically reduces its communication performance.
- The high cost of context switching on the CM-5 dramatically reduces the advantage of multithreading.

In addition, various other architectural features have an effect on system performance. First, the expense of active message support on Alewife makes our optimistic active messages mechanism (which is described in Section 6.2.3) perform poorly. Second, the register windows on the CM-5 reduce the performance of active messages, and are the primary reason that context switches are so expensive.

Section 4.1 and 4.2 describe the important characteristics of Alewife and the CM-5. The implementation of dynamic computation migration in MCRL on these two machines is described in Chapter 6. Section 4.3 summarizes the main differences between Alewife and the CM-5. Section 4.4 describes the Proteus simulator and the simulation of Alewife that we used in our work on static computation migration, which is described in Chapter 5.

4.1 Alewife

Alewife [1] is an experimental shared-memory multiprocessor. The Alewife architecture consists of processor/memory nodes connected by a packet-switched interconnection network. The network is organized as a low-dimensional mesh; on the current machine it has two dimensions. The raw bandwidth of an interconnection link is approximately 45–50 Mbytes / second in each direction.

Each processor / memory node consists of a Sparcle processor [2], an off-the-shelf floating-point unit, a direct-mapped, 64K unified cache with 16-byte lines, eight megabytes of DRAM, the local portion of the interconnection network, and a Communications and Memory Management Unit (CMMU). The 8M of DRAM is divided into three portions, the first two of which can be used by the programmer: 4M of shared memory, 2M of local memory, and 2M of shared-memory directory. The Sparcle processor is derived from a SPARC v7 processor, and as a result is similar to a CM-5 processor.

Alewife provides efficient support for both coherent shared-memory and message-passing communication styles. Shared memory support is provided through an implementation of the LimitLESS cache coherence scheme [21]. In LimitLESS, limited sharing of memory blocks (up to five remote readers) is supported in hardware; higher-degree sharing is handled in software by trapping the home processor.

In addition to providing support for coherent shared memory, Alewife provides the processor with direct access to the interconnection network [61]. Efficient mechanisms are provided for sending and receiving both short (register-to-register) and long (memory-to-memory) messages. Using these message-passing mechanisms, a processor can send a message in a few user-level instructions. A processor that receives such a message traps; user-level software can respond either by executing a message handler or by queuing the message for later consideration.

4.1.1 Hardware Overview

Alewife uses the SPARC register windows [86] as four hardware contexts for fast multithreading. As an example use for these contexts, the Alewife runtime system can be configured to automatically context switch to another hardware context upon a remote memory reference. The hardware contexts are also used for handling active messages; a hardware context is always left free for active messages to execute in. My experiments only use one hardware context for computation, and leave the remaining ones for active message handling.

Two bugs in the CMMU chip are worth mentioning. First, because of a timing conflict between the CMMU and the floating-point unit, codes that make significant use of floating-point computation must run at 20 MHz instead of the target clock rate of 33 MHz. Because of this, all Alewife experiments are run at 20 MHz. Second, in order to ensure data integrity when using the block transfer mechanism, it is necessary to flush message buffers from the memory system before using DMA. This overhead cuts the peak bandwidth of the block transfer mechanism from approximately 2.2 bytes / cycle (44 Mbytes / second at a 20 MHz clock rate) to roughly 0.9 bytes / cycle (18 Mbytes / second). Both of these bugs will be fixed in a planned respin of the CMMU.

Alewife uses a heuristic deadlock avoidance mechanism for the network. Whenever a processor detects that its send queue is full for "too long," it assumes that the network is clogged. The processor begins taking packets out of the network until it determines that the network is unclogged; when the network is unclogged, it then relaunches these packets. This mechanism definitely affects performance (and there appear to be bugs when network traffic is high), but turning it off risks deadlock. In my experiments the timeout is set to 65535 cycles (approximately 3.3 milliseconds).

4.1.2 Software Overview

The Alewife C compiler is a research compiler that achieves approximately 90% of the performance of `gcc` with an optimization level of `O2`. It compiles a superset of C that includes a set of message-passing mechanisms. The compiler and runtime system provide support for creating computation in the form of two calls: `thread_on` and `user_do_on`. The former call is used to create a thread (on any processor) to initiate computation; the latter is used to start an active message.

Threads on Alewife are non-preemptive. In other words, threads must explicitly yield the processor. However, a thread can be interrupted by an active message, which executes in a separate hardware context. It takes approximately 175 cycles for a thread to be created and then start running after the thread creation message interrupts the processor. This assumes that no other thread is running, and that a context switch to the newly created thread happens immediately.

Active messages in Alewife are more general than those originally described by von Eicken *et al.* [95]. Active messages run atomically; that is, other active messages cannot interrupt the processor while an active message is running. It costs approximately 60 cycles for an active message to start executing after it has interrupted a processor.

The Alewife runtime system allows active messages to become "half-threads." A half-thread can be interrupted by another active message, but cannot block. However, if the total number of active threads and half-threads overflows the number of free hardware contexts, a half-thread is descheduled. The transition to a half-thread is accomplished by the `user_active_global` call, which takes about 100 cycles in the fast case (when the instruction path is warm and a free thread exists). In the respin of the CMMU, the fast-case transition should only take a few cycles.

The support for the transition to a half-thread allows an active message to use software atomicity mechanisms. Before the transition to a half-thread, an active message is guaranteed to run atomically. During this time, it can setup software mechanisms for atomicity. Following the transition, it can then assume that it is still running atomically. Figure 4-1 illustrates this use of `user_active_global`; it contains an abstracted version of some message handling code from within the CRL distributed shared memory system.

The message handler illustrated in Figure 4-1 runs as an active message. It is not interruptible until it executes `user_active_global`. If the global variable `atomicity_bit` is set, it means that a previous handler is executing; a continuation for the current call must be created. Otherwise, `atomicity_bit` is set, and the call is executed.

The half-thread transition can be viewed as a variant of optimistic active messages [49, 97], which is describes in Section 6.2.3. Active messages on Alewife have hardware support in the form of the hardware contexts that Sparcle provides. However, the transition does not the same functionality as optimistic active messages. Several important restrictions of the current implementation of the runtime system limit its usefulness. These problems are not inherent in the system, but are merely artifacts of the current runtime system.

First, active messages can only transition to half-threads, not full-fledged threads that can block. Allowing optimistic active messages to block is one of the important features of the optimistic active message model. Future versions of the Alewife kernel should allow active messages to transition to full threads.

```
void message_handler(void)
{
  /* active message handler */
  if (atomicity_bit) {
    /* create a continuation
       and discard the message
     */
    return;
  }

  /* extract the function and its arguments
     from the message,
     and discard the message
   */

  /* set the software atomicity bit */
  atomicity_bit = 1;

  /* transition to a half–thread */
  user_active_global();

  /* execute the function */

  atomicity_bit = 0;
}
```

Figure 4-1. Software atomicity with half-threads on Alewife. This code is abstracted from the internals of the CRL distributed shared memory system.

Second, active messages and their associated half-threads are not guaranteed to execute in LIFO order. When an active message transitions to a half-thread, it may be descheduled to allow the last active message to run. Allowing a half-thread to be descheduled breaks the intended model of execution, where the transition allows further active messages to interrupt the processor. As a result, guaranteeing atomicity during the execution of nested active messages is expensive. This effect is discussed in more detail in Section 6.4.1.

The compiler and runtime system provide support for fast atomic sections through the use of a pair of calls called `user_atomic_request` and `user_atomic_release` [14]. The effects of these calls are similar to turning off and on interrupts, respectively. The calls block and unblock, respectively, incoming user-level active messages; system-level active messages can still execute. Both calls take only a few cycles to execute in the fast case; the latter call traps into the kernel if an active message arrived during the atomic section. This mechanism is similar to one suggested by von Eicken in his thesis [94]; it is also similar to a mechanism described by Stodolsky *et al.* [90].

4.1.3 Ideal Alewife

Some of my measurements are on an "ideal" Alewife machine. In particular, Section 7.2.2 describes the presumed costs of data migration and computation migration on such a machine. Those measurements assume the following two changes that will be in the respin of the CMMU:

- The DMA bug is fixed. As a result, the effective bandwidth of memory-to-memory transfers increases by a factor of 2.5, from about 18 Mbytes/second to about 45 Mbytes/second.

- The local-to-global transition for active messages only takes a few cycles.

These measurements are taken on the actual Alewife machine. The first difference is accounted for by not flushing any of the message buffers. Since my measurements on the "ideal" Alewife machine do not use of any of the data transferred, the effects of the DMA bug are not seen. "Real" code cannot be run on the "ideal" Alewife machine, because the DMA bug can corrupt data that is used.

The second difference is accounted for by not transitioning active messages to half-threads. Such an optimization is safe for simple measurements, because only one thread makes requests at a time. This optimization cannot be done safely for measurements of any programs that contain concurrency, since letting active messages run for long periods of time can deadlock the network.

The results for running on an ideal Alewife machine are measured with a clock rate of 20MHz, which allows for comparison with the measurements of the real machine.

4.2 CM-5

The Thinking Machines' CM-5 is a message-passing multicomputer based on a fat tree network [64]. Each processor has two network links, each with a bandwidth of 20 Mbytes/second each way. The maximum achievable total bandwidth is approximately 11.8 Mbytes/second [16], which is a limit imposed by the software overhead of handling messages.

My experiments were run on a 128-node CM-5 system that runs version 7.4.0 Final of the CMOST operating system and version 3.3 of the CMMD message-passing library [93]. Each CM-5 node contains a SPARC v7 processor that runs at 32 MHz, and 32 Mbytes of physical memory. Each node has a direct-mapped, 64K unified cache [94].

4.2.1 Hardware Overview

The CM-5 network has several properties that are unusual for multiprocessor networks. First, the interconnection network provides multiple paths between processors. As a result, message reordering is possible: messages sent between two processors can arrive out of order. Thus, the CM-5 looks much like a cluster of workstations, in which the network is commonly assumed to reorder messages.

Second, non-deterministic routing is used in the CM-5 network. As a result, the performance of the network can appear to vary widely under contention.

Third, the CM-5 network is context-switched along with the processors: in-transit messages are dumped to the processors. Context-switching the network and the processors together enables entire partitions and the network associated with them to be safely gang scheduled. Strict gang scheduling of the processors and the network allows user-level access to the network interface, because applications cannot send messages to one another. Unfortunately, draining the network during a context switch is known to cause measurement problems on the CM-5. Drainage of the network on timer interrupts appears to happen even in dedicated mode; this effect increases the variability of network performance.

The two network links into each processor are used as two networks in order to avoid deadlock. In the CMAML communication library these networks are used as request-response networks: requests go on one network, and replies go on the other.

Finally, the network only supports single packet messages; additionally, a packet consists of only five words. As a result, active messages [95] on the CM-5 are restricted in that they can have only four arguments. As a result, any communication that requires more than four arguments costs a great deal more. The CMAML library provides a bulk data movement facility called `scopy`; it uses hard-wired active messages to avoid sending a handler address in data packets.

4.2.2 Software Overview

The compiler we used on the CM-5 is gcc 2.6.3, with the optimization level set at `O2`. In CMMD terminology [93], the programs are node programs; that is, there is no master program running on the host.

The CM-5 was one of the platforms for the research on active messages [95]. Active messages reduce the software overhead for message handling: messages are run as *handlers*, which consist of user code that is executed on the stack of a running computation. Executing a handler on the current thread's stack avoids the overhead of thread management; a minimum-cost round-trip active message takes approximately 12 microseconds. Scheduling the execution of active messages is typically done by disabling and enabling interrupts, or by polling the network. Many applications choose to use polling, because the cost of changing the interrupt level is expensive on the CM-5.

The CM-5 does not provide any support for multithreading. The software environment is intended for SPMD programs, where each processor runs the same single-threaded program. This software model grew naturally out of SIMD machines such as Thinking Machines' CM-2, but is limited by the lack of multithreading.

A non-preemptive thread package is used to support multithreading on the CM-5. We developed this thread package for our work on optimistic active messages [97], which is described in Section 6.2.3. It is fairly well-tuned; as a baseline number, the fastest context switch time is approximately 23 microseconds.

The SPARC processor on the CM-5 has eight register windows that are managed as a ring buffer. One of the windows is kept invalid so as to mark the end of the buffer. As a result, six nested procedure calls can execute within the register windows. The seventh nested procedure call requires that at least one window be flushed to the stack. CMOST appears to flush a single window upon window overflow. The bottommost window is flushed to the stack, after which it becomes the invalid window; the invalid window then becomes valid. Upon window underflow, the reverse process occurs.

4.3 Discussion

There are several important differences between Alewife and the CM-5: communication performance, the costs for context switching, the presence of hardware contexts, and support for multiprogramming. The first two differences have an impact on my measurements, whereas the latter two do not.

The difference in communication performance has a large effect on the relative costs for data migration and computation migration, as will be shown in Chapter 7. The CM-5 is generally slower, except for active message handling. For a round-trip active message, the two machines have similar performance: about 12 microseconds for the CM-5, and about 15 microseconds on Alewife. However, there are several other factors that cause a large difference in communication performance:

- The time to transfer relatively short messages is much lower on Alewife. By "relatively short," I mean messages that are longer than four words, but not transferred using DMA on Alewife. On the CM-5, such messages are transferred using the `scopy` mechanism. One active message is required for every 4 bytes of payload, and an additional active message round-trip is required to set up an `scopy`. On Alewife, single messages that do not use DMA can contain up to 15 words of payload.

- The time to transfer large messages is higher on the CM-5. By "large," I mean messages that are transferred using `scopy` on the CM-5 and DMA on Alewife. The `scopy` mechanism on the CM-5 can achieve a maximum of 10 Mbytes/second [94], but requires an extra round-trip message to set up. The DMA hardware on Alewife can transfer data at 44 Mbytes/second, and does not require the processor to move the data. The DMA bug in the Alewife CMMU requires that message buffers be flushed from the cache, but Alewife's effective DMA rate of 18 Mbytes/second is still almost twice as high as the CM-5's transfer rate.

Context switching is much cheaper on Alewife. As a result, the performance of threads versus active messages on Alewife is much closer. There are several reasons why context switches on Alewife are relatively cheaper than on the CM-5:

- Only the registers for a single context need to be saved on Alewife, whereas the entire register set must be saved on the CM-5. As a result, the cost to save registers is much greater on the CM-5.

- During a context switch on the CM-5 there is no polling. As a result, active messages are blocked in the network for long periods of time on the CM-5. On Alewife interrupts are only disabled for a short time during a context switch.

The hardware contexts on Alewife, combined with the support for half-threads, can help active message performance; free contexts are used to handle active messages. However, this effect will not appear in my results, for two reasons. First, since active messages cannot transition to full threads, the cost of creating threads must still be paid if blocking can occur. Second, and more importantly, the active message support still contains some bugs at the time of writing.

Finally, Alewife is a single-user machine, whereas the CM-5 is a time-shared and space-shared multi-user machine. All of my experiments on the CM-5 are run in dedicated mode, so the effect of sharing the machine does not show up in my measurements.

4.4 Proteus and Simulated Alewife

Our initial research was performed on the Proteus simulator [15], an execution-driven parallel architecture simulator. Proteus delivers high performance by directly executing instructions rather than simulating them; simulator timing code is inserted into assembly code. As a result, the simulated processor architecture is the same as that of the architecture on which Proteus is run. The processor on which we ran Proteus is a MIPS R3000-based machine [56].

The machine architecture that we simulated was similar to Alewife, in that it used the LimitLESS cache coherence protocol [20]. In addition, each processor had a 64K shared-memory cache with a line size of 16 bytes. The primary differences between the Proteus model and Alewife are the following:

- The processor architecture is different. Proteus models a MIPS processor, whereas Sparcle is a SPARC-based processor with multiple hardware contexts.

- The network interface is different. Proteus does not model the memory-mapped register interface that Alewife provides.

- Local caching effects are not simulated. Because Proteus uses direct execution, the caching of instructions and local (non-shared) memory is not simulated.

We also performed experiments to estimate the effects of providing hardware support for user-level message passing. The addition of hardware support for RPC and computation migration makes the comparison to Alewife shared memory fairer, since shared memory has a substantial amount of hardware support. In our experiments we simulated a network interface that is integrated with the processor. In particular, we assumed that the network interface was mapped into ten additional registers in the register file, so that marshaling and unmarshaling would be cheaper. (We did not directly simulate the extra registers, but simply changed the accounting for the marshaling and unmarshaling costs.) The performance advantages of such an organization have been explored by Henry and Joerg [42].

Chapter 5

Static Computation Migration in Prelude

To evaluate the benefits of computation migration, we implemented static computation migration in the Prelude system. This chapter describes the Prelude implementation, and summarizes some of our performance results. A more complete description of our measurements and results can be found in our conference paper [50].

Our Prelude results lead to several conclusions:

- Computation migration, like data migration with replication, outperforms RPC. The reason is that both data migration with replication and computation migration improve the locality of repeated accesses.

- The performance of a software implementation of computation migration on a concurrent, distributed B-tree is half that of a hardware implementation of data migration with replication. This result demonstrates the usefulness of computation migration in improving performance, given the disparity in performance between hardware and software. It also demonstrates the need for a comparison between all-software systems.

- Hardware support for message-passing improves the performance of computation migration. On systems with very fast message-passing, computation migration can perform nearly as well as hardware data migration.

- Statically using computation migration for all remote accesses is insufficient. Performance could be improved by judiciously combining computation migration and data migration with replication.

Section 5.1 describes the implementation of static computation migration within Prelude. Section 5.2 describes some of our results, and Section 5.3 discusses the implications of these results.

5.1 Implementation

Prelude [100] is an object-based language that provides procedures and instance methods. Instance method calls are the remote accesses in Prelude, and procedures (single activations) can migrate to objects on which they invoke instance methods. Instance method activations are not allowed to migrate, because we require (in implementations that do not use shared memory) that all instance method calls run at the site of the invoked object. In shared-memory implementations, instance method calls "pull" copies of an object to themselves.

There are two parts to the Prelude implementation of computation migration. The first part of the implementation is how migration is expressed by the programmer using a simple annotation. The second part is how the compiler generates code for computation migration.

Prelude supports a simple program annotation, similar to that shown in Figure 3-2, that allows the programmer to indicate the use of static computation migration. The annotation is used to mark an instance method call as a migration point within a procedure; the procedure is migrated if the instance method call is non-local. Computation migration imposes no additional cost on local accesses, because every Prelude method invocation must check whether the invoked object is local.

We implemented computation migration within the compiler, as described in Section 3.1. The compiler, which generates C code, emits a "continuation" procedure at the point of migration. The continuation procedure's body is the continuation of the migrating procedure at the point of migration; its arguments are the live variables at that point. We used an imprecise, conservative live variable analysis for our experiments.

In order to handle successive migrations of computation, the compiler generates two continuation procedures at each point of migration. The first continuation procedure represents an initial migration. It is called when a procedure first migrates, and it returns its values to its caller through a return stub. The second procedure represents a later migration. It is called when the first continuation procedure migrates, and it returns its values to the first continuation's caller through its return stub. The linkage information for the first continuation's return stub is passed to the second continuation. When the first continuation procedure migrates (and, as a result, the second continuation procedure is called on another processor), it kills itself.

5.2 Prelude Results

One of our Prelude experiments was on a distributed B-tree implementation. This B-tree is a simplified version of one of the algorithms proposed by Wang [99]; it is described in more detail in Section 7.3.2. In our experiments, we first construct a B-tree with ten thousand keys; the maximum number of children or keys in each node is constrained to at most one hundred. The values of the keys range from 0 to 100,000, and the nodes of the tree are laid out randomly across forty-eight processors. We then create sixteen threads on separate processors that alternate between thinking and initiating requests into the B-tree. We ran experiments with two different think times, zero and ten thousand cycles. Each request had a 10% chance of being an insert, and a 90% chance of being a lookup.

Scheme	Throughput (operations / 1000 cycles)
data migration	1.837
RPC	0.3828
RPC w / hardware	0.5133
RPC w / replication	0.6060
RPC w / replication & hardware	0.7830
computation migration	0.8018
computation migration w / hardware	0.9570
computation migration w / replication	1.155
computation migration w / replication & hardware	1.341

Table 5.1. B-tree throughput in Prelude: 0 cycle think time

In our experiments, we compared RPC, data migration in the form of Alewife shared memory, and computation migration. Under RPC, each instance method call executed at the object upon which it was invoked. Under data migration, shared memory was used to pull objects to the site of the callee. Finally, under computation migration, the annotation described in the previous section was used to move the calling procedure to the object. We used computation migration on both read and write operations.

Tables 5.1 and 5.2 present the results of the experiments using a zero cycle think time. "Hardware" in the first column of the tables means that we simulated a register-based network interface, as described in Section 4.4. It is important to remember that we compared a software implementation of RPC and computation migration to a hardware implementation of data migration, namely Alewife-style cache-coherent shared memory. Cache-coherent shared memory benefits from hardware support in three ways. First, it provides cheap message initiation. Data migration messages are launched and consumed by the hardware most of the time. Second, it provides cheap global address translation, without which a global object name space must be built in software. Finally, it provides cheap read and write detection on cache lines.

In all of our B-tree experiments, computation migration has higher throughput than RPC, but lower throughput than data migration. In fact, even with hardware support for RPC and computation migration, the throughput of data migration is over three times that of RPC and almost twice that of computation migration. As explained in our paper [50], much of the relatively poor performance of computation migration and RPC can be attributed to the inefficiencies in the runtime system.

In addition, replication contributes significantly to data migration's throughput advantage. One of the limiting factors to B-tree performance is the root bottleneck; it is the limiting factor for RPC and computation migration throughput. Computation migration, for instance, moves an activation for every request to the processor containing the root. The activations arrive at a rate greater than

Scheme	Bandwidth (words / 10 cycles)
data migration	75
RPC	7.3
RPC w / hardware	9.9
RPC w / replication	7.0
RPC w / replication & hardware	9.3
computation migration	3.5
computation migration w / hardware	4.3
computation migration w / replication	3.8
computation migration w / replication & hardware	3.9

Table 5.2. B-tree bandwidth demands in Prelude: 0 cycle think time

the rate at which the processor completes each activation. Data migration, however, provides local copies of B-tree nodes to alleviate resource contention.

Although data migration alleviates resource contention, it requires a great deal more network bandwidth. The amount of network bandwidth required to maintain cache coherence can be seen in Table 5.2; maintaining coherence imposes a substantial load on the network. Furthermore, the cache hit rate was less than seven percent; we can conclude that data migration benefited from its caches only because of its replication of the root of the B-tree.

We expect computation migration's performance to improve considerably when software replication is provided for the root; as shown in Table 5.1, this expectation holds true. However, the throughput of computation migration is still not as high as that of data migration. This difference remains because we still experience resource contention; instead of facing a root bottleneck, we experience bottlenecks at the level below the root — the root node has only three children.

We then set up an experiment in which this new source of resource contention was alleviated. In this experiment, the B-tree nodes are constrained to have at most only ten children or keys (but all other parameters are identical). The resulting throughput for computation migration with root replication is 2.076 operations / 1000 cycles vs. 2.427 operations / 1000 cycles for data migration. While data migration still performs better, computation migration's throughput is better because the bottleneck below the root has been alleviated. The reason for this alleviation is twofold. First, the roots of the trees in these experiments have four (instead of three) children. Second, activations accessing smaller nodes require less time to service; this lowers the workload on the frequently accessed processors. From these experiments, we conclude that the main reason for data migration's performance advantage is the replication due to hardware caches. This result correlates with conclusions made by other studies examining B-tree performance and caching [29, 99].

Keeping the above information in mind, the throughput of computation migration should be closer to that of data migration in a B-tree experiment where contention for the root is much lighter

Scheme	Throughput (operations / 1000 cycles)
data migration	1.071
computation migration w / replication	0.9816
computation migration w / replication & hardware	1.053

Table 5.3. B-tree throughput in Prelude: 10000 cycle think time

Scheme	Bandwidth (words / 10 cycles)
data migration	16
computation migration w / replication	2.5
computation migration w / replication & hardware	2.7

Table 5.4. B-tree bandwidth demands in Prelude: 10000 cycle think time

(thus reducing the advantage provided by caching). Tables 5.3 and 5.4 show performance results for experiments conducted with a ten thousand cycle think time; for brevity, RPC measurements have been omitted. With hardware support, computation migration and data migration have almost identical throughput. Again, data migration uses more bandwidth because it must maintain cache coherence.

5.3 Discussion

In the two applications that we describe in our conference paper (one of which is described here), static computation migration always outperforms RPC. However, a hardware implementation of data migration with replication outperformed our software implementation of computation migration. Such a result is not surprising, given the speed of hardware relative to software. What is surprising is that the performance of the software implementation of computation migration approaches that of the hardware implementation of data migration with replication.

Our Prelude experiments with the B-tree showed that a static choice of always using computation migration is insufficient. In particular, the root of a B-tree should be replicated using data migration, and the leaves should be accessed using computation migration.

In conclusion, although our results on Prelude are promising for computation migration, they also illustrate the need for more research. First, they demonstrate that a better comparison would involve two software-based implementations of computation migration and data migration with replication. Such a comparison would remove any difference in hardware support for the two mechanisms. Second, they illustrate the need to explore the combination of computation migration and data migration. A judicious mixture of the two mechanisms could yield better performance than either could alone.

Chapter 6

Dynamic Computation Migration in MCRL

To evaluate the effectiveness of combining computation migration and data migration, the dynamic protocol described in Section 3.3.3 was implemented in the MCRL distributed shared memory system. MCRL is a multithreaded extension of CRL, a distributed shared memory library built at MIT [51]. MCRL extends CRL in two ways: first, it provides support for multithreaded programs; second, it provides support for dynamic computation migration.

Multithreading is important for several reasons. First, it is an important structuring tool. Many programs are naturally written with multiple threads of control. Second, it is a useful mechanism for hiding latency; in particular, the latency of remote accesses. Finally, for the purposes of evaluating computation migration, multithreading is a necessity. Moving computation from processor to processor makes sense only if the model of execution is multithreaded.

An important difference between our work in Prelude and the work in MCRL is that the MCRL system is all-software. As a result, it provides a "level playing field" with which to evaluate the dynamic computation migration protocol and the two decision heuristics. In addition, the performance results from MCRL have been measured from real machines, not a simulator. These results provide a sanity check on our simulated results, and also verify that we can achieve good performance on real machines.

A last difference between Prelude and MCRL is that MCRL does not have any compiler support. The CRL shared-memory interface is fairly high-level, and can be used by a programmer. However, it requires that the programmer manage the translation between global identifiers and local identifiers. The MCRL interface for dynamic computation migration is even more complex, and it is unlikely that a programmer would want to write for it.

However, the intent in using MCRL is not to treat it as a programmer interface. Instead, the goal is to treat it as a compiler target. Although providing simple annotations for computation migration is important, the primary goal of this dissertation is to demonstrate the usefulness of mechanisms for dynamic computation migration. Compiler mechanisms for providing annotations such as those described in Section 3.2 are important, but not the main focus in this dissertation.

In the remainder of this dissertation, the term “pure data migration” will mean the use of the standard CRL calls. That is, the programmer decides to always use data migration, and directly uses the CRL calls. I use “dynamic data migration” and “dynamic computation migration” to refer to the use of the dynamic protocol in MCRL. In other words, one of the heuristics is invoked at runtime, and the runtime system decides to use either data migration or computation migration.

The remainder of this chapter describes my implementation of dynamic computation migration in MCRL. Section 6.1 overviews the DSM interface that the CRL system provides. Section 6.2 describes high-level implementation issues that arise with computation migration and MCRL. Section 6.3 describes the differences between CRL and MCRL: first, the changes to support multithreading; and second, the changes to support dynamic computation migration. Section 6.4 describes some detailed implementation issues for the two machines that MCRL runs on: Alewife and the CM-5.

6.1 CRL Overview

CRL (short for C Region Library) is a C library that provides an interface for distributed shared memory [51]. Its global name space is in the form of *regions*, which are programmer-defined blocks of memory. Regions are identified by *region identifiers*, or rids, for short. These regions are analogous to objects in other parallel languages. Unlike hardware implementations of shared memory, software implementations depend on programmer intervention to define regions for coherence; only by doing so can the cost of checking for remote accesses in software be amortized over the size of an object.

CRL provides support for SPMD parallel programs. The programming model is that each processor runs a single thread. Whenever a thread makes a remote request, it spins until its request is fulfilled. Its model of coherence is that of entry consistency; the local copy of a region is only valid while an appropriate lock for that region is held. Locking happens implicitly with the access functions in CRL, which acquire regions in either read-mode (shared) or write-mode (exclusive).

Each region in CRL has a fixed home processor. A processor that is not the home for a particular region is a remote processor. The home processor acts as a server for a region; remote processors that access a region are clients of its home processor.

Table 6.1 lists the shared memory operations that CRL provides. The top half of the table contains operations that are global; they operate on rids. The bottom half of the table contains operations that are local; they operate on mapped regions. CRL also provides several global synchronization operations that are not listed. Technical details on the internal structure of CRL are given in Appendix A.

Region metadata is stored in a *region header*. Regions on both home nodes and remote nodes have headers, although the information stored in them is different. CRL divides the protocol for accessing a region into two phases. The first phase consists of mapping a region, which sets up the region header. The `rgn_map` operation translates an rid to a local address. It allocates memory to hold a region and its header, initializes the header, and returns a pointer to the region.

Function	Effect
Global operations	
`rgn_create`	create a new region
`rgn_delete`	delete a region
`rgn_map`	map a region locally
Local operations	
`rgn_unmap`	unmap a mapped region
`rgn_start_read`	start a read operation on a region
`rgn_end_read`	end a read operation on a region
`rgn_start_write`	start a write operation on a region
`rgn_end_write`	end a write operation on a region
`rgn_flush`	flush the local copy of a region
CRL constants	
`crl_self_addr`	local processor id
`crl_num_nodes`	total number of processors

Table 6.1. Summary of the CRL shared memory interface. The global synchronization operations are not listed.

The second phase of accessing a region consists of initiating a read or a write using the `rgn_start_read` or `rgn_start_write` procedures, respectively. When these procedures complete, they ensure that the region data is coherent. If the data is not local, the procedures send any appropriate messages and spin while waiting for responses. Once the data arrives, it is guaranteed to remain coherent until the matching termination procedure (`rgn_end_read` or `rgn_end_write`) is called. The `rgn_unmap` procedure can then be used to unmap a region, after which its local address is no longer valid.

CRL caches unmapped regions and their headers in a data structure called the *unmapped region cache*, or URC. As the name indicates, when a region is unmapped it is added to the URC, which is direct-mapped. If the region is in the URC and `rgn_map` is called on it, no communication is required to map it. Regions are only evicted from the URC upon a conflict. The version of CRL on which MCRL is based does not manage the URC very well; a more recent version of CRL achieves higher hit rates in the URC.

CRL has good performance, as shown in Tables 6.2 and 6.3; CRL on an ideal Alewife machine would perform within a factor of 15 of Alewife itself. The data in these tables is partially taken from the original version of the paper by Johnson *et al.* [52]. In these tables, the numbers marked "Alewife (ideal)" do not include the overhead of flushing message buffers and transitioning messages into threads. These ideal latencies represent the latencies that should be possible after the Alewife CMMU respin (which is described in more detail in Section 4.1). The CM-5 numbers in Tables 6.2 and 6.3 are slightly different than those in the original version of the CRL paper. These numbers

	CM-5		Alewife		Alewife (ideal)		Alewife (native)	
	cycles	μsec	cycles	μsec	cycles	μsec	cycles	μsec
start read hit	78	2.4	46	2.3	47	2.3	—	—
end read	91	2.8	50	2.5	50	2.5	—	—
start read miss no invalidations	1714	53.6	959	48.0	580	29.0	38	1.9
start write miss one invalidation	3374	105.5	1620	81.0	978	48.9	66	3.3
start write miss six invalidations	4936	154.2	3009	150.5	1935	96.7	707	35.4

Table 6.2. Measured CRL latencies, 16-byte regions (in both cycles and microseconds). Measurements for Alewife's native shared memory system are provided for comparison. Most of these numbers are taken from the original CRL paper [52].

		CM-5		Alewife		Alewife (ideal)	
		cycles	μsec	cycles	μsec	cycles	μsec
start read	miss, no invalidations	3635	113.6	1123	56.2	642	32.1
start write	miss, one invalidation	5308	165.9	1776	88.8	1046	52.3
start write	miss, six invalidations	6885	215.2	3191	159.6	2004	100.2

Table 6.3. Measured CRL latencies, 256-byte regions (in both cycles and microseconds). Most of these numbers are taken from the original CRL paper [52].

```
extern void f1(object a);        extern void f1a(local_object *);
extern void f2(object a);        extern void f2a(local_object *);
extern void f3(object a);        extern void f3a(local_object *);

void foo(object a)             void foo(rid_t a)
{                               {
  f1(a);                          local_object *a0 = rgn_map(a);
  f2(a);                          rgn_start_read(a0)
  f3(a);                          f1a(a0);
}                                 f2a(a0);
                                  f3a(a0);
                                  rgn_end_read(a0);
                                  rgn_unmap(a0);
                                }
```

Figure 6-1. Hand optimization of code in MCRL. The program fragment on the left-hand side represents code in a high-level language. The program fragment on the right-hand side represents code in MCRL. The procedure `f1` has the same effect as `f1a`; similarly for the procedures `f2` and `f2a`, and `f3` and `f3a`.

are measured on a newer version of the CMOST operating system, as well as a newer version of the CMMD library.

CRL currently runs on the MIT Alewife machine, on Thinking Machines' CM-5 machine, and on a network of Sun workstations.

6.2 Implementation Issues

This section describes issues that arise in the use of dynamic computation migration and MCRL. Section 6.2.1 describes some of the language issues involved with using a low-level object model. Section 6.2.2 describes related compiler issues. Section 6.2.3 overviews our research on optimistic active messages, which is an important mechanism for making communication efficient.

6.2.1 Language Issues

Some of the locality issues that arise in higher-level languages such as Prelude do not arise in MCRL. In particular, since the programmer is responsible for managing the translation of region identifiers to local addresses, the programmer will also optimize code around these translations. In other words, the programmer will tend to hoist global address translation out of procedure calls. As a result, some abstractions will take global addresses as arguments and others will take local addresses. Figure 6-1 illustrates this effect.

In Figure 6-1 the program fragment on the left-hand side represents code in a high-level language that provides an address space of objects, such as Prelude. Invocations are performed on global addresses, and abstractions such as `f1`, `f2`, and `f3` are written to take global addresses as arguments. The fragment on the right-hand side represents code in a system such as MCRL. The programmer is responsible for translating global addresses into local addresses, and naturally hoists such translations out of procedures (assuming that releasing locks between the procedure calls is not necessary for correctness). For my experiments I wrote code in the latter form. Such a coding style represents the best performance that a compiler for a high-level language could achieve by hoisting locking and global address translation out of procedures and loops.

6.2.2 Compiler Issues

An annotation similar to that described in Section 3.2 could be used to indicate the use of dynamic computation migration. Handling an annotation requires either changes to the C compiler, or the addition of a preprocessing phase. Since our current implementation is done in the context of MCRL and the C language, there would be several language restrictions that were not present in Prelude. Note that if MCRL were implemented only on shared-memory multiprocessors, these restrictions would be unnecessary.

First, if the address is taken of a stack location, the containing activation frame cannot be migrated. If the frame were migrated and the address were stored in the heap, the address would be a dangling reference. Second, if the address of a stack location is passed to a procedure and not ignored, the activation frame for the procedure cannot be migrated. The stack location is a local address, and cannot be used on another processor.

In addition, the C compiler must understand MCRL's data types. In particular, it must understand the distinction between region identifiers and local pointers. Local pointers cannot be passed off-node; only the rid that corresponds to a local pointer can be passed to another processor. For every pointer that needs to be migrated, the compiler must know to send the corresponding rid instead.

Finally, the compiler must understand the semantics of MCRL locking. If a computation were to migrate while it holds a lock, the ownership of that lock must be passed along with the computation. The compiler would have to understand this semantics, and handle it correctly. The current version of MCRL does not support the migration of lock ownership, but it would not be difficult to add such support.

6.2.3 Optimistic Active Messages

In other work [49, 97] we have demonstrated the usefulness of optimistic active messages. Optimistic active messages are a generalization of active messages. Active message handlers must be restricted because of the primitive scheduling control available for active messages. Optimistic active messages, on the other hand, allow more general message handlers: there are fewer restrictions on code that can be put in message handlers.

Active message handlers are generally run as interrupt handlers; this fact, combined with the fact that handlers are not schedulable, puts severe restrictions on the code that can be run in a message handler. For example, a message handler is not allowed to block, and can only run for a short period of time. Active messages that block can create deadlocks; long-running handlers can congest the network. In our papers [49, 97] we describe how using optimistic active messages eliminates these restrictions. Optimistic active messages allow arbitrary user code to be written as message handlers. They also allow arbitrary synchronization between messages and computation; in addition to this gain in expressiveness, optimistic active messages will generally perform as well as active messages.

Optimistic active messages achieve the performance of active messages by allowing user code to execute in a message handler instead of a thread. By executing in a handler, we avoid thread management overhead and data copying time. Because handlers are not schedulable entities, executing arbitrary user code in message handlers is difficult. As mentioned above, handlers may not block, and can only run for a short period of time. Our solution is to be *optimistic* and compile handlers under the assumptions that they run without blocking and complete quickly enough to avoid causing network congestion. At runtime, these assumptions must be verified.

If our optimistic assumptions fail at runtime, we “abort” the optimistic active message. When an optimistic active messages aborts, we revert to a slower, more general technique for processing the handler. Possibilities include creating a separate thread or putting an entry on a job queue.

This optimistic approach to active message execution eliminates thread management overhead for handlers that run to completion, and will achieve performance equal to active messages when most handlers neither block nor run for too long. Furthermore, it frees the programmer from the burden of dealing with the restrictions of active messages, which greatly simplifies writing parallel applications.

My experiments make extensive use of optimistic active messages. Since we do not have compiler support for optimistic active messages, all of my optimistic active message handlers are hand-coded. My OAM abort code, which creates a thread upon abort, is also hand-coded.

6.3 MCRL Implementation

This section describes the implementation of MCRL. Section 6.3.1 describes the support for multithreading in MCRL, and the changes to CRL that were necessary. Section 6.3.2 describes the support for dynamic computation migration in MCRL.

6.3.1 Multithreading Support

CRL does not provide any facility for multithreading. It supports a SPMD model of computation, in which each processor runs exactly one thread. This section describes the changes necessary to make CRL multithreaded.

Although spinning while waiting for a lock or a message is efficient when there is only one thread per processor, it makes poor use of the processors when there are threads waiting to run. The major change in adding multithreading to CRL is to add blocking when most messages are sent. In addition, on Alewife I had to modify the kernel and sundry synchronization libraries; on the CM-5, I had to modify our user-level thread package. Although more complex strategies are

Function	Effect
`rgn_map_nonblock`	map a region without blocking (if it is cached or already mapped), otherwise return NULL
`rgn_start_read_nonblock`	start a read without blocking (if it is cached), otherwise return NULL
`rgn_start_write_nonblock`	start a write without blocking (if it is cached), otherwise return NULL

Table 6.4. MCRL non-blocking operations. These operations are for use within active messages and optimistic active messages.

possible (for example, competitively spinning and then blocking, which has been investigated for synchronization strategies [58]), I chose to implement a pure blocking strategy for simplicity.

On Alewife, the fastest round-trip time for a remote request is roughly 22 microseconds. Thus, a context switch allows useful work to happen despite the cost of a context switch, which is slightly over 7 microseconds. On the CM-5, the fastest round-trip time for an active message is about 12 microseconds, whereas a context switch costs at least 23 microseconds. As a result, some fast remote calls do not block, as the latency of a fast call cannot be hidden by context switching.

Another change was to make multiple `rgn_map`'s on a processor return the same region. In MCRL, when a thread tries to map a remote region, it blocks. If another thread attempts to map the same region, it also blocks and waits for the result of the first `rgn_map`. This optimization saves both network bandwidth and memory; it ensures that only one message is sent, and ensures that only one copy of the region is mapped.

A series of other changes was necessary to support execution of MCRL calls within optimistic active message handlers. Supporting multithreading on Alewife and the CM-5 only requires the prevention of race conditions within voluntary yields of the processor, since neither machine currently has support for the preemption of threads. However, once MCRL calls can occur in active messages, there are many more race conditions to prevent, and the system begins to look preemptive.

Finally, several non-blocking operations had to be added in order to execute within optimistic active messages. As described in Section 6.2.3, blocking is not permitted within active messages. As a result, any call that can potentially block must have a non-blocking form as well. Table 6.4 lists the non-blocking calls in MCRL.

6.3.2 Dynamic Computation Migration Support

This section describes how the dynamic migration protocol is implemented in MCRL. It first describes the interface for dynamic computation migration, and then describes the implementation of the dynamic protocol.

Function	Effect
Client operations	
rgn_start_local_read	starts read if data is local and readable
rgn_end_local_read	used to end read if computation migrated
rgn_start_local_write	starts write if data is local and writable
rgn_end_local_write	used to end write if computation migrated
rid_home	return home processor of region
crl_thread_id	return id of current thread
rgn_migrated	check if region migrated
rgn_home_addr	return address of region at home
Server operations	
migration_policy_read	decide whether to migrate a read
migration_policy_write	decide whether to migrate a write
rgn_map_migrated	map a region at the home node
rgn_notify_rmigrate	notify client that migration happened on a read
rgn_notify_wmigrate	notify client that migration happened on a write
rgn_notify_rmigrate_and_kill	notify client that migration happened on a read, and kill waiting thread
rgn_notify_wmigrate_and_kill	notify client that migration happened on a write, and kill waiting thread
rgn_fake_read_msg	act as if a shared request has arrived
rgn_fake_write_msg	act as if an exclusive/modify request has arrived
Other	
rgn_header	return address of a region's header

Table 6.5. MCRL operations for dynamic computation migration

Interface

Table 6.5 lists the operations in MCRL that support dynamic computation migration. These operations are not intended for use by a programmer, but by a compiler — just as my intent is to use MCRL as a compiler target.

Figures 6-2 and 6-3 illustrate the use of these calls for dynamic computation migration. The client first calls `rgn_start_local_read`. This operation behaves identically to `rgn_start_read` if the data is local; otherwise, it changes the state of the region's metadata so that the processor can send a migration request. The client must check the return code of `rgn_start_local_read`, which indicates whether the region was local.

If the region is non-local, the client then initiates the dynamic protocol. It calls the continuation routine `server_read`, which is hand-written. The arguments `crl_self_addr`, `x`, `crl_thread_id()`, and `rv` are used for linkage. `rgn_home_addr(x)` and `crl_self_addr` are used internally by MCRL for data migration requests. After sending a migration request message, the client suspends. The argument to `crl_suspend` is used to detect the race condition where a wakeup returns before the thread suspends; if a wakeup does return early, `crl_suspend` merely returns.

The argument to `crl_suspend` is also used to indicate whether the server returned data or executed the computation. After waking up, the client uses `rgn_migrated` to check this flag. If the data is returned, the client executes the computation; otherwise, it falls through.

On the server side, the first action is to decide whether to use data migration or computation migration. This decision is made in the `migration_policy_read` call. If the server decides to use data migration, it executes `rgn_fake_read_msg`, which mimics the effect of handling a request for data. Otherwise, if the server decides to use computation migration, it returns a migration acknowledgment using `rgn_notify_rmigrate`. It then maps the region using `rgn_map_migrated`, which is faster than `rgn_map` because it does not need to check whether it is executing at the home node. Finally, it starts a read, executes the computation, and returns a value using `remote_wakeup`. This last call is not in the MCRL interface: it is written by the user (or generated by the compiler), because its interface depends on the number of return values.

Two of the calls listed in Table 6.5 require extra explanation, as they are not used in the example code. These calls are `rgn_notify_rmigrate_and_kill` and `rgn_notify_wmigrate_and_kill`. These calls are used in server code that is called as a result of a repeated migration. As described earlier, a repeated migration can return to the original caller. The above calls allow server code to kill an intermediate thread in a chain of migrations. Killing such a thread avoids the cost of having to reschedule the thread and then execute it, when all it will do is exit. Just as importantly, it reclaims thread resources; if threads are not killed, the number of threads can quickly overflow any limit.

The dynamic migration protocol is more expensive for writes than for reads. This difference is due to the fact that writes must manage version numbers. More specifically, when a client wants to acquire a write copy of a region of which it already has a read copy, it must send a message that includes the version number of the copy that it has. Although it would be possible to modify the MCRL interface to have two sets of write calls (one for a pure write and one for a write when a read copy is present), such a design would result in a more complex interface.

Finally, one early design choice was to use computation migration only for region data. An alternative choice would be to use computation migration for both region headers and region data. In other words, a call such as `rgn_map_local_and_read` could be added that maps and starts a read for a local region, or initiates computation migration. However, such a design would break MCRL's model, which is to split `rgn_map` and `rgn_start_read` into two operations.

Implementation

Dynamic migration can only occur on remote processors. Operations that occur at a region's home processor do not migrate, even if the only valid copy of the data is currently on a remote processor. Allowing home operations to migrate in such cases could lead to "migration thrashing," where a computation chases data from processor to processor. For example, if a home operation could migrate, it could migrate to a remote processor while the data is returning to the home; it could then migrate back to the home, and repeat the whole cycle again.

```
void client_read(rid_t r1)
{
  void *x;
  int migrate;
  unsigned notify;
  int migrateReturnValue;
  int *rv = &migrateReturnValue;

  x = rgn_map(r1);
  migrate = rgn_start_local_read(x, &notify);

  if (migrate) {
    /*
       thread_on is the Alewife call to create a thread on
       some processor
     */
    thread_on(rid_home(r1),          /* home processor   */
              server_read,            /* function to call */
              rgn_home_addr(x),       /* arguments        */
              crl_self_addr,
              x,
              crl_thread_id(),
              rv);
    crl_suspend(&notify);
    if (rgn_migrated(notify)) {
      rgn_end_local_read(x);
      goto done;
    }
  }

  /* COMPUTATION goes here */

  rgn_end_read(x);
 done:
  rgn_unmap(x);
}
```

Figure 6-2. Client code for dynamic computation migration

```
void server_read(HomeRegion *home_x,
                 unsigned proc,
                 void *remote_addr,
                 tid t,
                 int *rv)
{
  void *x;

  if (! migration_policy_read(home_x, proc)) {
    rgn_fake_read_msg(home_x,
                      proc,
                      rgn_header(remote_addr));
    return;
  } else {
    rgn_notify_rmigrate(proc, remote_addr);
  }

  x = rgn_map_migrated(home_x);

  rgn_start_read(x);

  /* COMPUTATION goes here */

  /* wakeup sleeping process

     user_do_on is the Alewife call to create an
     active message on a processor
   */
  user_do_on(proc,
             remote_wakeup,
             t,
             rgn_header(remote_addr),
             rv,
             1);
  rgn_end_read(x);
  rgn_unmap(x);
  return;
}
```

Figure 6-3. Server code for dynamic computation migration

Migration can occur in three cases: when a read misses, when a write misses, and when a write hits on a read-only copy. These situations correspond to the three types of remote requests: read requests, write requests, and modify requests. If a region is in any other state (for example, a read request is outstanding), my protocol blocks.

The state diagrams for CRL and MCRL are given in Appendix A. For simplicity, only one outstanding potential migration per region per processor is allowed. That is, if a thread attempts to migrate on a region while another thread on the same processor has sent a migration request on that region, the second thread blocks until a migration acknowledgment or data is received.

Blocking the second thread in the above situation avoids another form of migration thrashing. For example, in response to the first potential migration the home could send data back to the remote processor. If the second request did not block, but instead sent another potential migration to the home, the advantage of having sent the data back is lessened.

In the implementation of my protocol, the extra acknowledgment for a migration adds an additional message send to the latency of the migration, but the reception is off of the critical path of the migration. As soon as a thread decides to use computation migration, it sends the migration acknowledgment. It would be possible to send this acknowledgment later; for example, after the computation is in the process of potentially migrating to another processor. However, delaying the acknowledgment could cause potential migrations to "pile up" at the client.

The STATIC heuristic does not require any information in a region header, as it is a static protocol. The REPEAT protocol requires four words in a home region's header to store 128 bits of information; these bits are used to detect when a remote processor accesses a region twice. These four words are sufficient to represent up to 128 processors.

6.4 Implementation Details

This section describes implementation details of the two platforms on which MCRL currently runs: the MIT Alewife machine and Thinking Machines' CM-5. Neither machine supports pre-emptive scheduling; as a result, threads must yield the processor explicitly.

In MCRL writers are given priority when threads can be woken. In other words, when a writer finishes it does not wake up any readers unless there are no writers waiting. Although this bias is not fair to readers, it reduces the chance that writers will starve.

The largest difference between Alewife and the CM-5 is the nature of the runtime systems. Alewife is largely interrupt-driven; interrupts and traps are relatively inexpensive on Alewife, due to the presence of hardware contexts. However, the `user_atomic_request` and `user_atomic_release` calls, which "shut off" user-level active messages, allow the programmer to use a form of polling when necessary. In comparison, since interrupts are expensive on the CM-5, applications almost always use polling to obtain higher performance.

6.4.1 Alewife

The Alewife implementation of MCRL uses the support for multithreading that the Alewife system provides. It was necessary in some cases to add some additional support for blocking in the Alewife kernel and libraries. Even though the active message support on Alewife is analogous to optimistic active messages, the restriction that half-threads cannot block means that the active message support is not much better than that on the CM-5. Besides the issues described in the rest of this section, at the time of writing there still appears to be some bugs in the active message support. As a result, the application performance results do not include times for optimistic active messages.

The Alewife compiler produces efficient code, but it has some quirks that affect the results of my experiments. In particular, all registers are caller-save. The reason is that the back end is shared between the C compiler and the Mul-T compiler. Mul-T [60] is a dialect of Lisp for parallel computation. Since Mul-T allows the creation of continuations and closures, a caller-save protocol is most sensible. For C code, however, a caller-save protocol can lead to unnecessary movement of data back and forth from the stack.

It was necessary to change the scheduling of threads in the Alewife runtime system. The original system executes threads in LIFO order: newly created threads are put on the front of the scheduling queue. This scheduling strategy is patently unfair, and can lead to some distorted measurements. In particular, as discussed in Appendix B, an unfair strategy can lead to overestimating the throughput of the system. As a result, the scheduler was changed so that threads are created in FIFO order. In addition to avoiding the aforementioned measurement effects, a FIFO discipline also matches the scheduling strategy on the CM-5.

The Alewife runtime system has extremely primitive scheduling, and for simplicity I avoided making any other changes to the scheduler. As a result, my heuristics for dynamic computation migration do not make use of load balance information, which may be useful. The effect of having such information would be useful to explore.

Optimistic Active Messages

As a result of problems with Alewife's active message support, the cost of using optimistic active messages on Alewife is high; using threads turns out to be faster. Therefore, all of the application measurements on Alewife are taken with user code running as full threads, and not as active messages. The remainder of this section describes the difficulties with active messages on Alewife, which are artifacts of the current implementation of the kernel.

Ignoring the bugs in the active message support, the restrictions on active messages are still sufficient to make the use of optimistic active messages ineffective. As a result of the non-LIFO ordering of half-threads, guaranteeing message atomicity efficiently is difficult. MCRL uses a single bit to maintain atomicity in software; this bit is turned on and off around critical sections. However, because non-LIFO execution of half-threads is possible, a stack discipline cannot be maintained for this bit. In other words, an executing thread or optimistic active message cannot assume that the bit remains off once it has turned it off. There are two solutions to this: an optimistic active message can keep the bit locked for its entire execution; alternatively, an optimistic active message can recheck the bit whenever it needs atomicity, and abort if the bit has been turned on. (These options are roughly analogous to forward and backward validation in optimistic concurrency control systems [41].) My implementation uses the first option for simplicity.

Keeping the atomicity bit on means that more optimistic active messages are likely to abort. Abort is expensive for several reasons. First, my implementation of abort for optimistic active messages uses the Alewife `thread_on` call to create a thread locally. Unfortunately, `thread_on` sends a message to the processor on which a thread is to be created even when the processor is local. Second, starting an active message and transitioning to a half-thread costs almost as much as starting a thread. These implementation artifacts have the effect of degrading abort performance on Alewife: an aborting optimistic active message must pay the cost of starting an active message, sending and receiving a message, and creating a thread.

Finally, half-threads that overflow the number of hardware contexts will run slowly. For example, an active message can interrupt the processor and transition to a half-thread; if the half-thread is in the last free context, it is unloaded. The previous half-thread continues execution, and if another active message arrives it will also be unloaded upon transition.

6.4.2 CM-5

The CM-5 implementation of MCRL uses a user-level thread package that we developed for our work on Optimistic RPC [97]. The thread package requires that interrupts be off during calls to thread operations, which is how it achieves atomicity. Since toggling the interrupt mask on the CM-5 is expensive (it is on the order of hundreds of cycles [94]), all of the experiments described in this dissertation are with interrupts turned off; polling is used to receive messages.

A context switch in our thread package takes approximately 23 microseconds. Most of this time is the cost to save and restore two register windows, which is about 20 microseconds. The context switch cost also depends on the number of active register windows. Based on the calling conventions in the thread package, there are a minimum of two active windows: the window for the current procedure, and the window for the context switch code. There is an additional cost of approximately 5 microseconds for each extra window that must be saved. In addition, each window that must be restored after a context switch takes about 5 microseconds.

The SPARC register windows also impact procedure call time. Each nested procedure call after the seventh is expensive. Each extra procedure call takes approximately 10 microseconds. When the call stack overflows the register windows, it takes about 5 microseconds to push the bottom activation record onto the stack; when that activation record is returned to (and the call stack underflows the register windows), another 5 microseconds must be paid to load it. Under non-multithreaded uniprocessor workloads, register window handling only accounts for about 3% of the total cycles [86]. However, the cost for using register windows is higher for active messages (and optimistic active messages), because active messages execute on the stack of the currently executing thread, which deepens the call stack.

The largest factor in the performance of my dynamic migration protocol is the limited message size on the CM-5. The messages in my protocol are longer than four words, whereas the standard CRL calls only send messages of four arguments, ignoring region data transfers. As a result, code that uses pure data migration only requires a single round-trip active message for all of its requests. Computation migration, on the other hand, requires an extra round-trip active message to transfer its extra arguments, which are sent using the CMAML `scopy` facility.

6.5 Limitations

This section describes some of the limitations of MCRL, and discusses how they can be addressed.

My implementation of dynamic computation migration suffers from several drawbacks. First, it leads to a large expansion in the size of the code: for each occurrence of computation migration, I have a continuation procedure. Second, each of these continuation procedures is hand-generated; it would be much easier if I had compiler support. Third, all of the optimistic active message code is also hand-written; the necessity for code to handle optimistic active message aborts leads to further code expansion.

A compiler with support for computation migration could avoid generating the extra continuation procedure. In addition, with support for optimistic active messages it could also avoid generating extra abort procedures. At the point of migration any live variables would be marshaled into a message, which would be sent to the destination processor. At the destination processor, an alternate entry point to the migrating procedure would be called, which would unmarshal the values into an identical activation record, and would then jump back into the original procedure's code. The only tricky point would be if the migrating procedure was originally expected to return values on the stack; we would have to ensure that the code on the destination processor returns to a stub that returns a message to the original processor.

One limitation of MCRL is that it has not been heavily tuned. Although it is fairly efficient, there are several aspects of its performance that could be improved. For example, the version of CRL upon which MCRL is based manages the unmapped region cache poorly; incorporating recent CRL changes into MCRL would improve its performance. In addition, region headers are fairly large (they are approximately 45 words), which is a large space overhead for small objects.

Finally, a weakness of MCRL is that the coherence protocol that it implements does not have any mechanisms for "short-circuiting" data transfers from one remote processor to another. For example, in a counting network a processor that writes a balancer must often acquire the data from a remote processor. Under the coherence protocol implemented in MCRL, the data is first sent to the home processor, and then back to the requesting processor. Modifying the protocol to allow the data to be sent directly to the requesting processor (which is done, for example, in the DASH protocol [65]) would improve the relative performance of data migration.

Chapter 7

Performance

This chapter discusses and analyzes the performance of MCRL, as well as the performance of dynamic computation migration under my two heuristics.

Several important results regarding computation migration are demonstrated:

- First, MCRL has good performance, which makes it a good platform for comparing data migration and computation migration. MCRL is compared both to CRL and to the Alewife machine.

- Computation migration of writes improves performance, as demonstrated by a microbenchmark and two application kernels. The STATIC heuristic almost always outperforms pure data migration.

- Dynamic computation migration of reads can sometimes improve performance. The REPEAT heuristic almost always outperforms pure data migration, and sometimes outperforms the STATIC heuristic.

Section 7.1 describes how the performance results in this chapter were computed; it also discusses several other measurement details. Section 7.2 analyzes low-level measurements of MCRL: operation latencies, migration costs, and a microbenchmark. Section 7.3 analyzes measurements of two application kernels. Section 7.4 summarizes the results in this chapter.

7.1 Measurement Techniques

This section describes the measurement strategy used in this chapter, as well as some measurement issues. The technique of taking a linear regression of multiple measurements is the same as that used in the original CRL paper [51].

My results compute the average time for a single iteration of some experiment. Some of the experiments are single-processor, whereas most involve multiple processors. For the former, a single iteration means the time for the processor to complete the operation (or set of operations) in question. For the latter, a single iteration means the average time for all of the processors to complete the operation.

The average time per iteration is measured by taking a linear regression of data from multiple runs of each experiment. (The number of runs, and the number of iterations in each run, are described separately for each experiment.) The linear regression is of data of the form (i, t), where i is the number of iterations performed in the run, and t is the time the run takes. The computed slope of the least-squares line is the average time for a single iteration. Error bars in my graphs represent 95% confidence intervals on the slope. Appendix C summarizes the relevant regression formulas.

In order to factor out cache differences as much as possible, dynamic computation migration and pure data migration are compared using the same executable. That is, dynamic computation migration under the STATIC executable is not compared to pure data migration under the REPEAT executable, and vice versa. This keeps the difference due to cache effects at a minimum when comparing results. However, in order to allow for some comparison between the two heuristics, I pad the region headers in MCRL for the STATIC heuristic so that they are the same as for the REPEAT heuristic. This minimizes cache effects due to different region sizes and alignments.

Finally, it is necessary to factor out the cost of remote `rgn_map` operations. The design of the MCRL interface deliberately avoided using computation migration for `rgn_map`; therefore, including the cost of `rgn_map` calls that require communication would make computation migration appear to perform badly. In order to factor out these costs, the unmapped region cache is made fairly large. In addition, all of my experiments first map all of the regions on all of the processors, which warms up the URC. As a result, the URC miss rate is low.

It is important to note that measurements on the CM-5 were all taken in dedicated mode. In non-dedicated mode, the timers provided by the CMMD library can be inaccurate, even when there are no other programs running. Although there can be some timing inaccuracies even in dedicated mode, the inaccuracies are substantially lower than in non-dedicated mode.

As can be seen in the results, consistent timings are difficult to acquire on the CM-5. This effect is a result of two factors. First, the context switching of the network can lead to timing error, as described above. In addition, the non-deterministic routing scheme used in the network can also cause timing differences.

Finally, the left y-axes of all of the graphs in this section are times in microseconds. The graphs in each section are at the same scale to allow for comparison of the relative performance of Alewife and the CM-5. The right y-axes give times in cycles for the respective machines.

7.2 Low-level Measurements

This section analyzes three sets of low-level measurements of MCRL. First, it analyzes the raw performance of MCRL by comparing it to CRL, as well as comparing the performance of MCRL on Alewife to Alewife itself. It then compares the costs of data migration and computation migration in MCRL. Finally, it uses a simple microbenchmark to measure the efficacy of the STATIC and REPEAT heuristics.

		CM-5		Alewife		Alewife (ideal)	
		cycles	μsec	cycles	μsec	cycles	μsec
start read	hit	66	2.1	70	3.5	70	3.5
end read		90	2.8	77	3.9	80	4.0
start read	miss, no invalidations	1995	62.3	1170	58.5	845	42.3
start write	miss, one invalidation	2950	92.2	1980	99.0	1238	61.9
start write	miss, six invalidations	4163	130.1	3410	170.5	2189	109.4

Table 7.1. Measured MCRL latencies, 16-byte regions (in both cycles and microseconds). These measurements correspond to those in Table 6.2 on page 56.

		CM-5		Alewife		Alewife (ideal)	
		cycles	μsec	cycles	μsec	cycles	μsec
start read	miss, no invalidations	4049	126.5	1402	70.1	888	44.4
start write	miss, one invalidation	4998	156.2	2157	107.8	1286	64.3
start write	miss, six invalidations	6202	193.8	3611	180.6	2200	110.0

Table 7.2. Measured MCRL latencies, 256-byte regions (in both cycles and microseconds). These measurements correspond to those in Table 6.3 on page 56.

7.2.1 Operation Latencies

This section analyzes the costs for MCRL operations. The measurements of MCRL indicate that MCRL is efficient, and that as a result it is a good platform for evaluating dynamic computation migration.

The costs of each operation are measured by taking the regression of 64 runs, where the ith run consists of performing the operation on i regions. Tables 7.1 and 7.2 list some of the costs for operations in MCRL. These costs correspond to the CRL costs listed in Tables 6.2 and 6.3 (on page 56).

Local Operations

Local operations (a start read that hits, or an end read) in MCRL are in general more expensive than in CRL, because MCRL operations can potentially block. CRL operations do not block; they spin and wait for their effects to complete.

For a start read, the possibility of blocking means that the operation might have to retry itself. As a result, it must check a return code, which takes a few extra cycles. In addition, a few cycles must be paid to setup a loop for retrying the operation. For an end read, the possibility of blocking means that the operation may issue some wakeups: if there are any waiting writers, an end read must wake them up. At the minimum, a few cycles must be spent checking for waiting writers.

The ideal Alewife costs are virtually identical to the non-ideal costs for local operations. The presence of the DMA bug does not affect local operations; nor does the expense of transitioning local threads.

Finally, the relative overhead of MCRL versus the Alewife machine is greatest for local operations. Hardware and operating system implementations of distributed shared memory, such as Alewife and Ivy, respectively, do not impose any extra costs on local operations. MCRL, however, requires that start read and end read operations must be called, which together take approximately 150 cycles.

Remote Operations

Remote operations in MCRL on Alewife are several hundred cycles more expensive than the same operations in CRL. The primary cost is that incurred by context-switching a waiting process; even if there is no other runnable thread, the Alewife runtime system will unload a thread that suspends, and then load it back when it wakes up.

Some of the remote operations on the CM-5 are actually cheaper in MCRL than in CRL. For example, the write misses on the CM-5 are cheaper. The reason is that MCRL uses polling to receive active messages, whereas the version of CRL measured by Johnson *et al.* uses interrupts. In these experiments only one active thread exists in the system, which is the one that initiates the MCRL operations. The other processors simply sit in the scheduler and poll the network. As a result, messages are received more quickly than in the CRL experiments, since the overhead of interrupts on the CM-5 does not have to paid.

The remote operations on the CM-5 are not as expensive as one would expect. Given that a context switch costs 23 microseconds, it might seem unusual that the MCRL cost for a remote read is only 12 microseconds slower than in CRL. The reason is that the thread package on the CM-5 does not perform a context switch if the last thread to run becomes the next thread to run. In these experiments, since there is only one thread that suspends and wakes up, no context switches need to occur. In a situation where multiple threads are runnable, the cost of a full context switch would have to be paid.

MCRL on Alewife compares well to Alewife hardware. For remote operations that only involve hardware on Alewife (a remote read that causes no invalidations, or a remote write that causes one invalidation), the Alewife system is approximately twenty times faster than MCRL on Alewife. This relative performance is good for software, considering that software emulation of hardware is often two orders of magnitude slower than the hardware itself. For remote operations that require software intervention on Alewife (a remote write that invalidates six readers), the Alewife system is only three times faster than MCRL. The primary reason for the remaining difference in performance is that messages such as invalidations and invalidation acknowledgments are consumed in hardware.

7.2.2 Migration Latency

This section compares the costs of data migration and computation migration in MCRL on Alewife and on the CM-5; in addition, it compares these costs on an ideal Alewife machine. The results demonstrate how the size of data affects the relative costs of moving data and computation on each architecture. The results also demonstrate how Alewife is much faster than the CM-5 in moving data, because it allows for DMA from the network interface.

Three different costs are compared: the cost of using data migration under the CRL interface, the cost of dynamically using data migration, and the cost of dynamically using computation migration. The cost to choose whether to use data migration or computation migration is not measured in these experiments. For computation migration, the cost is for a null computation: no computation is migrated. Thus, the measurements are equivalent to measuring the cost of null RPC using the dynamic protocol. These measurements only measure the relative overhead of colocating data and computation.

The costs are measured by taking the regression of 64 runs, where the *i*th run consists of performing the operation on *i* regions. For data migration, the cost includes mapping a region, locking it in read mode, unlocking it, and unmapping it. For computation migration, the costs includes mapping a region, sending a migration message, waiting for it to return, and unmapping the region.

Alewife

Figures 7-1 and 7-2 illustrate the relative costs of data migration and computation migration on Alewife. For all of the curves, the cost is for a read operation; a write using the dynamic protocol is slightly more expensive.

The optimistic active message implementation of the dynamic protocol is approximately 4% faster than the thread version. This speedup is because the thread version must go through the scheduler, whereas active messages execute in a new hardware context. Nevertheless, even though the optimistic active message implementation is faster for a single migration in isolation, it is slower in general. This reversal is because of the Alewife limitations on active messages that are described in Section 6.4.1.

Figure 7-1 illustrates the cost of accessing one region on Alewife with either data migration or computation migration. The costs for computation migration and data migration are virtually identical for tiny regions. However, as the size of the region increases, computation migration begins to outperform data migration. The cost for computation migration stays constant, since it is independent of the region size; the cost for data migration increases linearly in the region size. At 2048 bytes, computation migration is 111% faster than data migration.

Figure 7-2 illustrates the costs of data migration on Alewife. Pure data migration is approximately 7% percent faster, since it does not incur the overhead of the dynamic protocol. Although it is not immediately evident from the graph, the costs for data migration are piecewise linear; at approximately 500 bytes the marginal cost per byte for data migration increases. At smaller sizes the incoming DMA to receive the data is overlapped with protocol computation; at larger sizes the computation must wait for the DMA to complete.

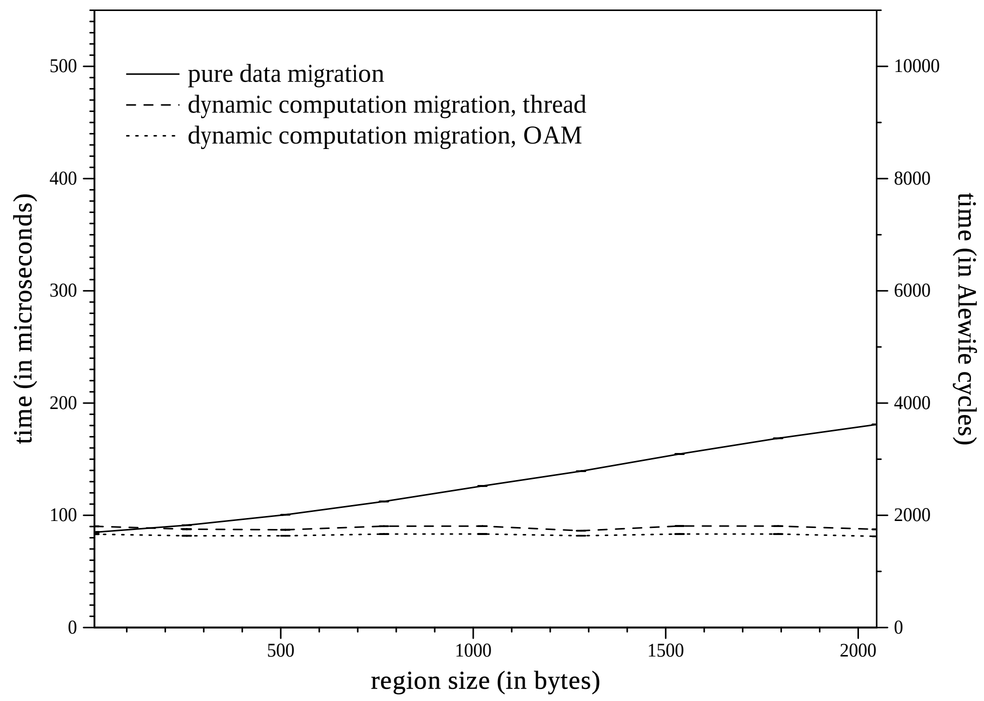

Figure 7-1. Comparison of data migration costs and computation migration costs for MCRL on Alewife

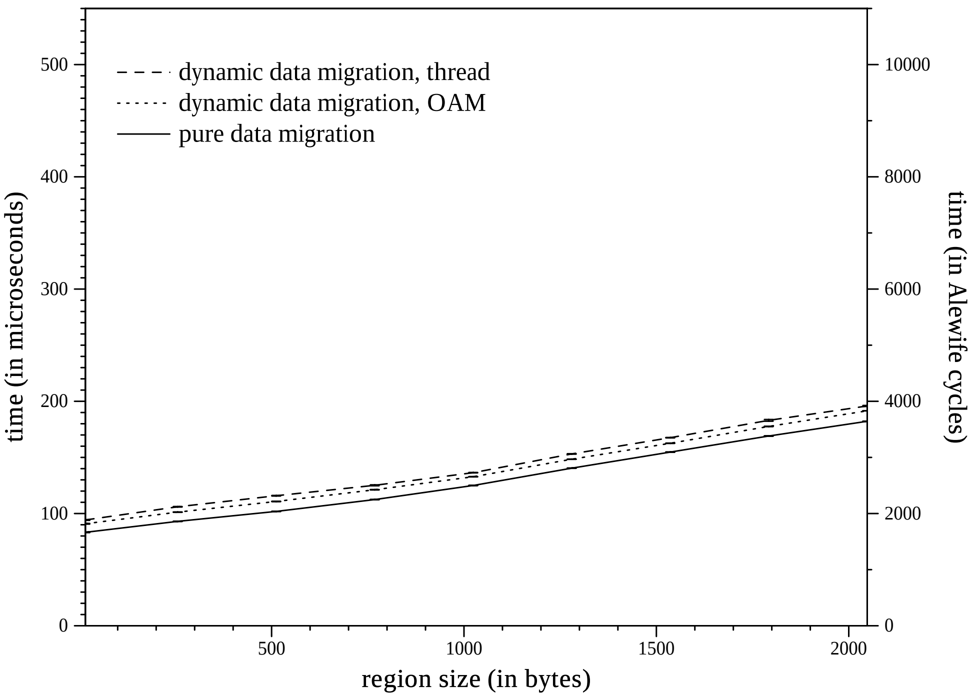

Figure 7-2. Comparison of data migration costs for MCRL on Alewife

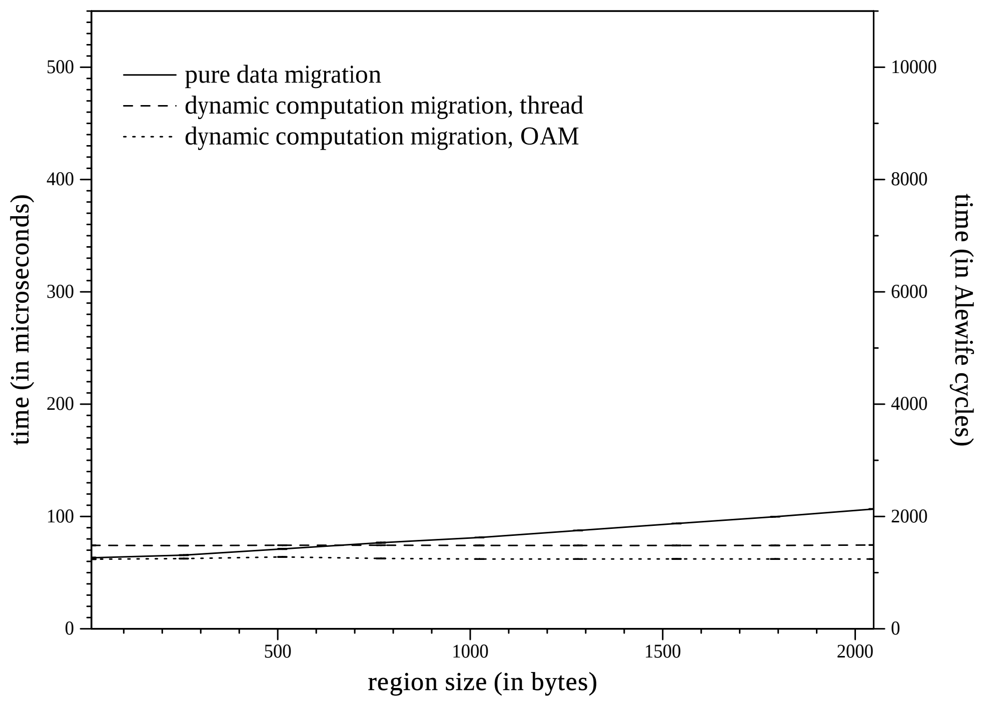

Figure 7-3. Comparison of data migration and computation migration costs for MCRL on an ideal Alewife machine

Ideal Alewife

Figures 7-3 and 7-4 compare the costs for pure data migration, dynamic data migration, and dynamic computation migration on an ideal Alewife machine. That is, measurements were taken without flushing message buffers or transitioning active messages to half-threads, as described in Section 4.1.3. These optimizations are possible because the measurements do not depend on the correctness of the data that is transferred, and because there is no concurrency in the experiment. These graphs demonstrate how faster communication reduces the effect that data size has on the performance of data migration.

As the graphs illustrate, the CMMU changes will decrease the performance gain of computation migration. For regions of 2048 bytes, computation migration is only 71% faster on an ideal CMMU; this speedup compares with an 111% performance difference on the real CMMU. The improvement in the local-to-global thread transition improves the performance of both data migration and computation migration. The major difference comes from fixing the DMA bug. This fix makes memory-to-memory transfer more efficient, which increases the performance of data migration. However, although the quantitative results of using dynamic computation migration would change, the qualitative results should not change dramatically.

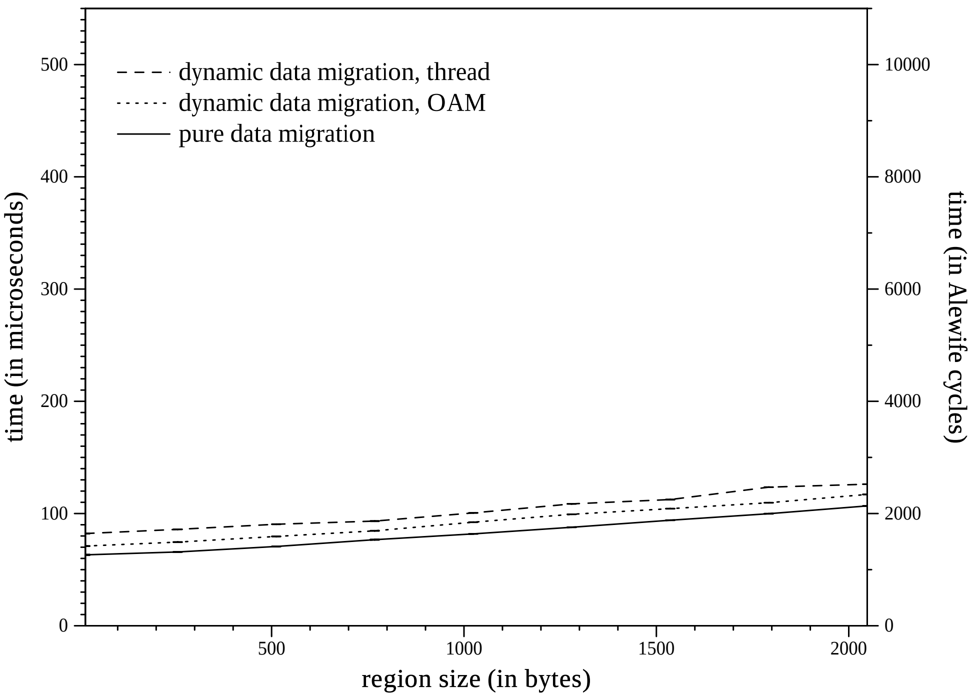

Figure 7-4. Comparison of data migration costs for MCRL on an ideal Alewife machine

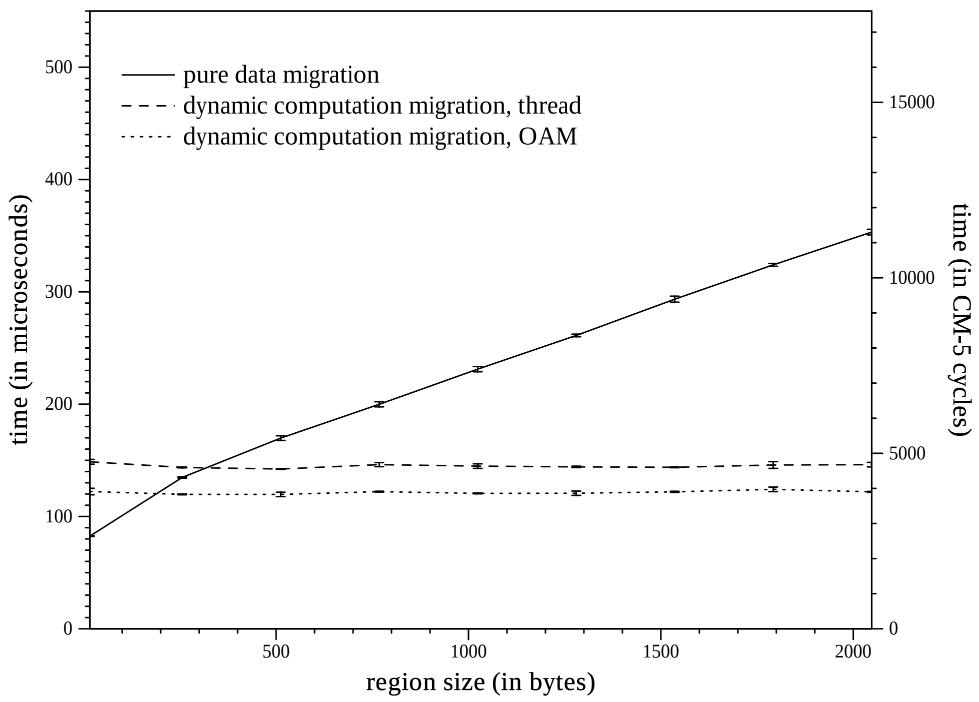

Figure 7-5. Comparison of data migration and computation migration costs for MCRL on the CM-5

CM-5

Figures 7-5 and 7-6 illustrate the corresponding costs on the CM-5. The wide error bars are due to unavoidable noise in the measurements. Using the dynamic protocol is much more expensive than using pure data migration, as an extra round-trip message is required. As a result, data migration is always significantly slower using the dynamic protocol. However, for large data sizes, computation migration still outperforms data migration (both pure data migration and dynamic data migration).

The curves in Figure 7-6 are piecewise linear. The breakpoint occurs because MCRL does not use the CMAML `scopy` primitive for data transfers of less than 384 bytes. Instead, it uses a specialized active message protocol to transfer data. This specialized protocol is faster than `scopy`, but it only sends three data words per packet. The `scopy` primitive sends four data words per packet by not sending a handler address: it uses a hard-coded active message handler to receive data [94]. That is why the curves in Figure 7-6 flatten out after 384 bytes: `scopy` sends fewer messages per byte than the specialized protocol.

The thread implementation is always slower than the active message implementation. Most of the extra time is spent in context switching. For computation migration (Figure 7-5), and for small data transfers for data migration (Figure 7-6), the cost is approximately 23 microseconds. The cost

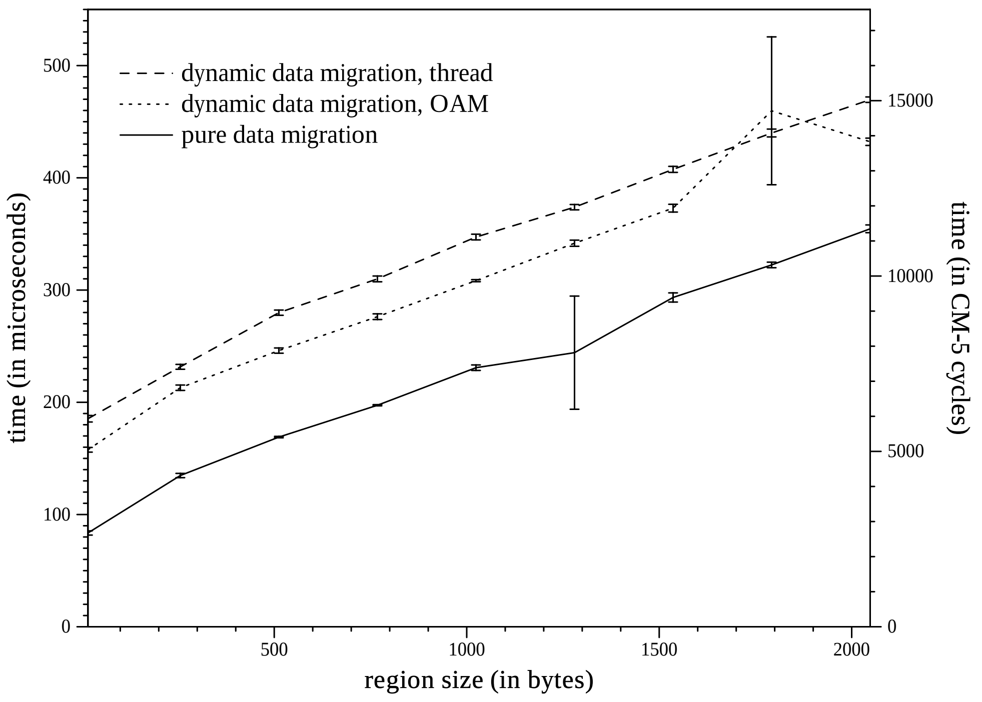

Figure 7-6. Comparison of data migration costs for MCRL on the CM-5

of a single context switch to the newly created thread accounts for this time. The second context switch away from the thread is not on the critical path.

For large transfers of data (regions larger than 384 bytes), the cost for using threads increases to 45 microseconds. The reason is that the second context switch is on the critical path. The server node must send a round-trip active message to the client in order to initiate an `scopy`. However, when the active message returns, the server is in the middle of the second context switch. No polling occurs in the middle of the context switch. As a result, the `scopy` does not begin until after the second context switch completes.

The major overhead of the dynamic protocol is the slowness of dynamic data migration. This overhead is approximately 70 microseconds, and is due to several factors. The extra round-trip message to setup an `scopy` is approximately 34 microseconds: roughly 12 microseconds for an active message, plus 23 microseconds for the context switch on the server. The `scopy` itself takes approximately 24 microseconds. A penalty of 5 microseconds must be paid because of one extra procedure call (and resulting window overflow) that the dynamic protocol currently uses. The corresponding window underflow cost is not on the critical path, and the overflow could be removed with inlining.

Discussion

The CM-5 is significantly slower than Alewife, in terms of both latency and bandwidth. Again, there are two primary reasons for this disparity in performance: the software overhead for moving data on the CM-5, and the limit on message length on the CM-5.

For large data transfers (those over 500 bytes), the data transfer rate on the CM-5 is approximately half that of Alewife: approximately 9 Mbytes/second versus 18 Mbytes/second. On an ideal Alewife machine, the data transfer rate would be even higher, at about 41 Mbytes/second. The difference in bandwidth is due to several factors; the primary factor is that Alewife uses DMA, where the processor must move the data on the CM-5.

An interesting point to note is that the active message performance on Alewife and the CM-5 is comparable. The cost to migrate 16-byte regions is roughly equivalent on the CM-5 and Alewife. On the CM-5, four active messages are used to move the data. On Alewife, a single message is used to DMA the region; since the overhead to receive an active message is higher on Alewife, the resulting latency is comparable to that on the CM-5.

7.2.3 Microbenchmark

This section describes results from a simple eight-processor microbenchmark. This microbenchmark compares the cost for data migration with replication against the costs of computation migration and RPC in the presence of contention. My protocol for dynamic computation migration is used to choose between moving computation and data. The results illustrate how increasing the rate of writes increases the cost of coherence, and how my heuristics can be used to reduce these costs.

The microbenchmark measures how well each heuristic does across a range of read/write mixes on a single 256-byte region. This region size is sufficiently small such that the cost of moving data is not a dominant cost. Each processor starts a single thread, which executes some number of iterations of a simple loop. In this loop, a processor generates a random number and decides whether to access the region in read or write mode. It then accesses the region in the appropriate mode, releases it, and then yields the processor. The cost of one iteration was measured by taking the regression of 20 runs, where the *i*th run consists of every processor executing $i \times 100$ iterations.

The results demonstrate that my two heuristics for choosing between computation migration and data migration can reduce the costs of coherence. The STATIC heuristic performs well when there are mostly reads or mostly writes, and generally outperforms data migration. The REPEAT heuristic performs better than the STATIC heuristic when the numbers of reads and writes are approximately equal, but performs slightly worse than the STATIC heuristic when there are mostly reads or mostly writes.

Alewife

The think time on Alewife is approximately 230 cycles: it takes approximately 55 cycles for the to generate a random number operation, and 175 cycles to compute modulus (it requires a system call, since Sparcle does multiplication and division in software).

STATIC heuristic Figure 7-7 illustrates the behavior of the STATIC heuristic. At 0% reads, every operation is migrated. The latency is that for always using computation migration. At 100% reads, every operation has data sent back. The latency is that for always using data migration with replication. The curve for the heuristic bulges upward. As the number of reads increases, data must be moved between the readers and the home. However, once reads begin to predominate, the performance advantage of replication becomes more of a factor. That is, when there is a sufficient percentage of reads such that caching is effective, the cost begins to decrease.

The STATIC heuristic saves some of the data transmission that pure data migration requires. In particular, data does not have to be moved to handle writes. Since all writes migrate to the home, an operation that follows a write does not need to move the data back through the home.

Although it is not immediately evident from Figure 7-7, at a high percentage of reads (near 100%) the heuristics perform a few percent worse than pure data migration. This performance degradation might be expected for the REPEAT heuristic, since it uses computation migration for some reads. However, this performance loss should not be that great for the STATIC heuristic, since it never migrates reads.

It turns out that near 100% reads, each iteration of the microbenchmark costs approximately 20 cycles more under the STATIC protocol. These cycles are lost on a local read; that is, when `rgn_start_local_read` is called when a read copy is present. Of these 20 cycles, 15 are artifacts of the compiler. Only four extra cycles should be necessary in this scenario. These cycles consist of the following steps:

- 1 cycle is necessary to load a stack address into an argument register; this argument is passed to `rgn_start_local_read`.

- 3 cycles are necessary to check the return value from `rgn_start_local_read`.

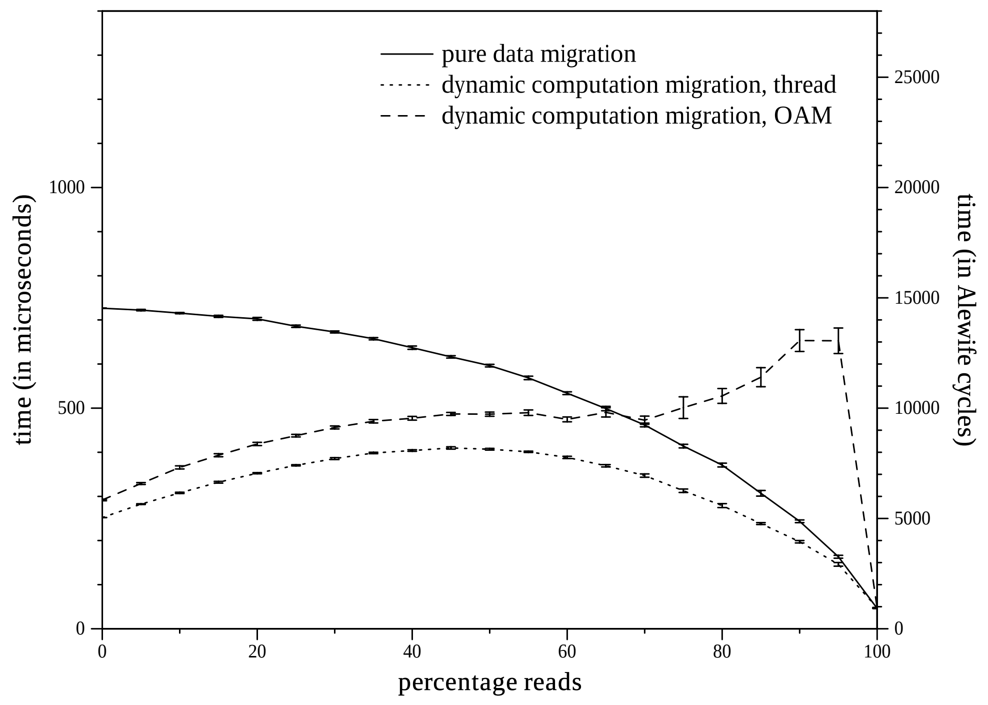

Figure 7-7. Cost per microbenchmark iteration under the STATIC heuristic on Alewife. In this experiment 8 processors access a 256-byte region.

The remaining cycles are due to artifacts of the Alewife C compiler. Most of the cycles are lost because the compiler makes all registers caller-save. As a result, there are several register saves and restores that are unnecessary. Another extra load is due to poor common subexpression elimination: a common subexpression that takes one cycle to compute is stored onto the stack and reloaded upon use. Finally, one instance of tail recursion optimization does not occur because the compiler does not know that a stack address is only used to pass a value back from `rgn_start_local_read` — this stack address is the one mentioned above.

The optimistic active message implementation is slower than the thread implementation. This slowdown is due to the two factors described in Section 6.4.1: optimistic active messages must run atomically by locking out other active messages (optimistic or not), and the global thread transition is expensive. It is not clear why the OAM curve has a peak at approximately 90% reads.

REPEAT heuristic Figure 7-8 illustrates the behavior of the REPEAT heuristic. Again, as for the STATIC heuristic, it is not clear why the OAM curve has a peak at approximately 90% reads. The REPEAT heuristic performs similarly to the STATIC heuristic at the endpoints: 0% and 100% reads. However, it avoids the upward bulge in the corresponding curve for the STATIC heuristic. This is

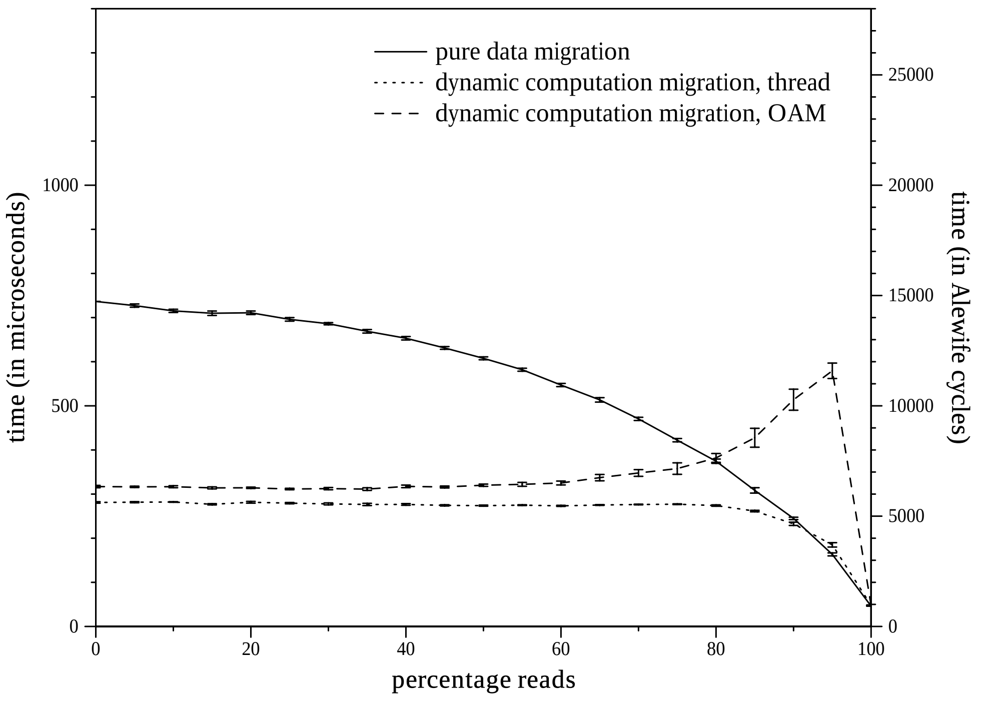

Figure 7-8. Cost per microbenchmark iteration under the REPEAT heuristic on Alewife. In this experiment 8 processors access a 256-byte region.

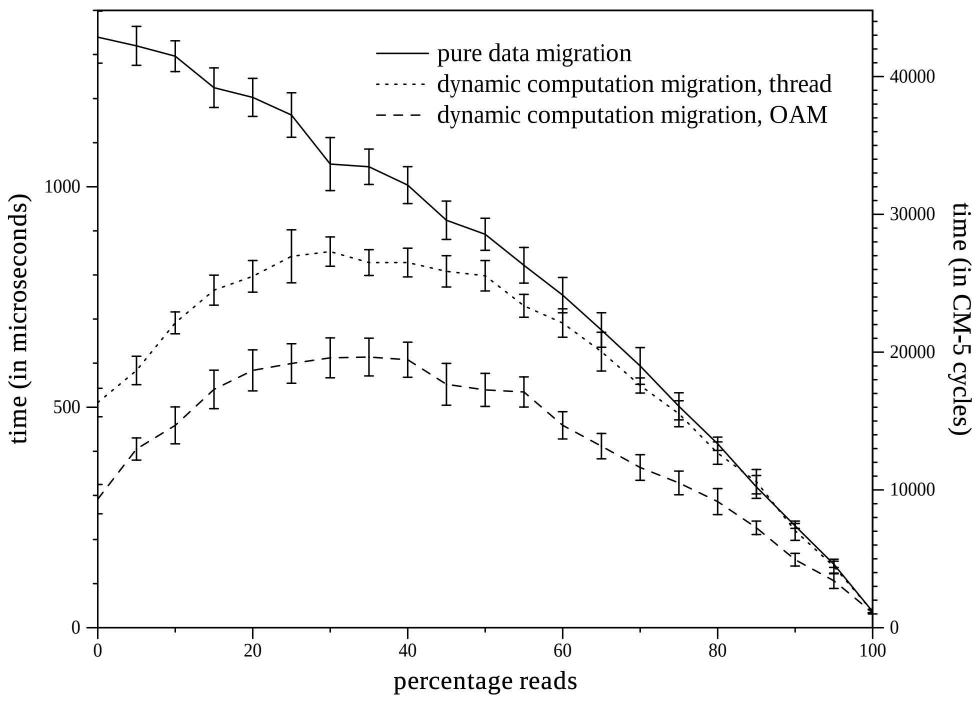

Figure 7-9. Cost per microbenchmark iteration under the STATIC heuristic on the CM-5. In this experiment 8 processors access a 256-byte region.

because it uses computation migration for reads when there is a high percentage of writes. All data transfer costs are eliminated, so the cost remains constant.

The REPEAT heuristic chooses to use data migration for reads when there is a large percentage of reads. Such a decision is sound, since data migration with replication is cost-effective when caching is effective. Caching is effective when there are repeated read accesses to the same data, which is exactly what the REPEAT heuristic detects.

CM-5

The costs (in cycles) for computing a random number and performing a modulus should be comparable to those on Alewife (roughly 55 and 175 cycles, respectively); the absolute time for the operations should be roughly 33% less, since the CM-5's clock speed is 50% higher than Alewife's.

STATIC heuristic Figure 7-9 illustrates the average latency per iteration for the microbenchmark on the CM-5 for the STATIC heuristic. As in the Alewife implementation, the curves for the dynamic protocol have minima at 0% and 100% reads. At 0% reads, computation migration is used exclusively; at 100% reads, data migration is used exclusively. As the number of reads increases from 0%, the cost increases because readers must have copies of the data migrated to them.

The difference between the thread and OAM implementations of dynamic computation migration is accounted for almost entirely by the cost of context switching. The thread implementation is approximately 260 microseconds slower per iteration, which is about 11 context switches (at 23 microseconds per context switch). This result implies that there are half of the context switches are "extra" context switches. In other words, since there are eight active threads on the home processor (one local and seven that are created in response to computation migration), approximately half go from one migration thread to the next. The other half of the context switches go from one migration thread back to the local thread.

REPEAT **heuristic** Figure 7-10 illustrates the behavior of the REPEAT heuristic. As in the Alewife implementation, the curve for the dynamic heuristic is flat over much of the domain of read percentages. In these operating ranges, the heuristic correctly chooses to use computation migration for reads and writes. As a result, the cost of moving data to readers does not have to be paid. As the number of reads approaches 100%, the heuristic chooses to use data migration, which allows readers to access cached copies of the data.

Discussion

The results for the STATIC heuristic indicate that statically deciding to use computation migration for writes and data migration for reads generally performs better than static data migration. Using computation migration for writes avoids the need to move data; in addition, it exploits the serial semantics of writes. Objects that are mostly written or mostly read are good candidates for using the STATIC heuristic.

The results for the REPEAT heuristic indicate that there is a benefit to dynamically choosing between computation migration and data migration for reads. In particular, objects that are accessed with a near-equal mix of reads and writes are good candidates for using the REPEAT heuristic.

The REPEAT heuristic performs better than the STATIC heuristic over the domain of read percentages. However, the STATIC heuristic performs better for strongly read-shared data. In addition, the STATIC heuristic performs nearly identically to the REPEAT heuristic for strongly write-shared data. The REPEAT heuristic only performs better when data has a near-equal mix of reads and writes.

The curves for the two heuristics have the same relative shape on both Alewife and the CM-5 (ignoring the bump in Figure 7-7). However, the time per iteration differs dramatically: the CM-5 is significantly slower. The reason is that the costs of both data migration and computation migration are greater on the CM-5.

The cost for data migration is roughly a factor of two greater on the CM-5. In the graph, this effect can be seen at 100% reads, where the cost for pure data migration is dominated by the cost for migrating data. This result matches the results given in Section 7.2.2. By comparing Figure 7-2 on page 75 and Figure 7-6 on page 79, we can confirm that the cost for migrating a 256-byte region on the CM-5 is roughly twice as expensive as on Alewife.

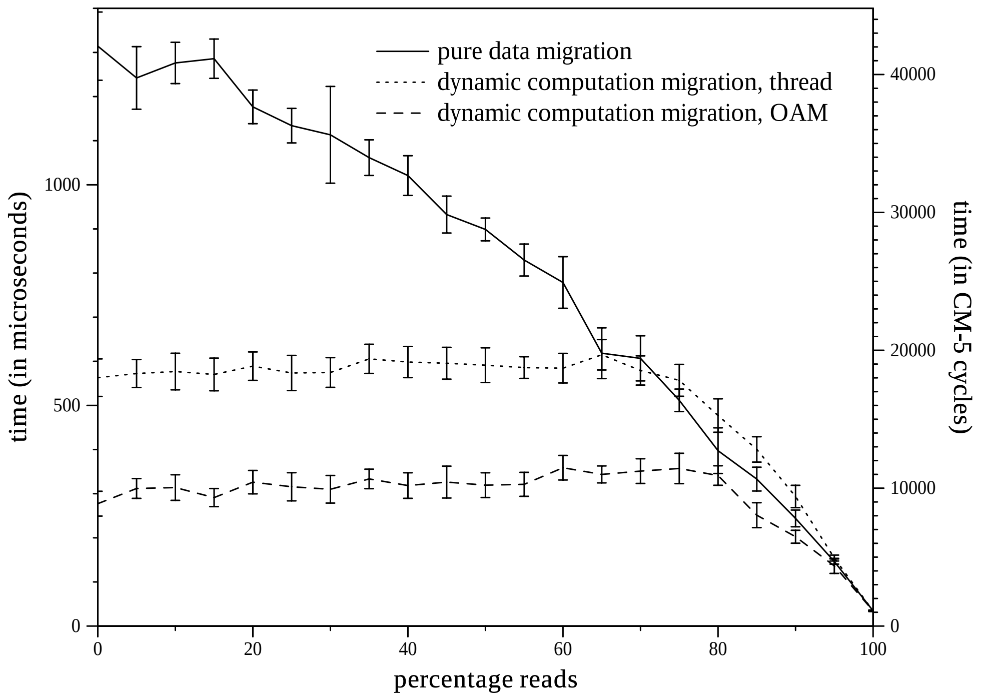

Figure 7-10. Cost per microbenchmark iteration under the REPEAT heuristic on the CM-5. In this experiment 8 processors access a 256-byte region.

Computation migration performs relatively better on the CM-5 than on Alewife. At 0% reads, dynamic computation migration is 2.9 times faster than pure data migration on Alewife; it is 4.6 times faster than pure data migration on the CM-5. The reason is that data migration is significantly slower on the CM-5, because the processor must move the data from the network interface to memory.

The variance of measurements on the CM-5 is much greater than on Alewife, because of several factors. First, since routing is non-deterministic on the CM-5, network traffic patterns can vary from run to run. Second, the timers provided by the CMMD library apparently can account for context switching incorrectly. In particular, the timers may measure time spent in context-switching the network, which appears to occur even in dedicated mode [16].

7.3 Application Kernels

This section analyzes measurements of MCRL on two application kernels. Results are presented for these two data structures, which are a counting network and a concurrent distributed B-tree. Both of these applications involve graph traversal, although their read / write characteristics are very different.

7.3.1 Counting Network

A counting network is a distributed data structure that supports "shared counting", as well as producer / consumer buffering and barrier synchronization [4, 43]. Shared counting occurs when multiple threads request values from a given range. For example, when a set of threads executes the iterations of a parallel loop, each iteration should be executed by exactly one thread. A shared counter allows threads to dynamically obtain iteration indices to execute.

The simplest data structure for shared counting is a counter protected by a lock: a thread acquires the lock and updates the counter when it needs a loop index. This solution scales poorly because of contention; a counting network is a distributed data structure that trades latency under low-contention conditions for much higher scalability of throughput.

A counting network is built out of balancers; a balancer is a two-by-two switch that alternately routes requests between its two outputs. Several $O(lg^2(n))$ depth counting networks have been discovered [4]: one is isomorphic to Batcher's bitonic sorting network, and the other is isomorphic to a balanced periodic sorting network. Figure 7-11 contains a diagram of an eight-input, eight-output bitonic counting network.

Experiments

The experiments were performed on an eight-by-eight (eight input, eight output) bitonic counting network. Figure 7-11 contains a diagram of such a counting network. The counting network is a six-stage pipeline. Each stage consists of four balancers in parallel, as shown in Figure 7-11. The counting network is laid out on twenty-four processors, with one balancer per processor. Placing the balancers on separate processors maximizes the throughput of the counting network. Each row

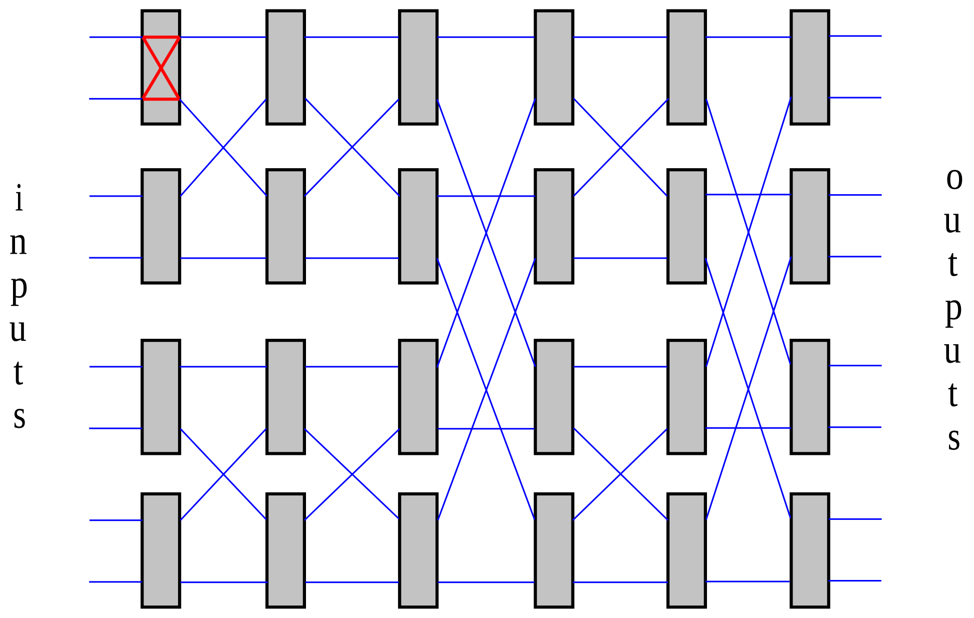

Figure 7-11. An eight-input, eight-output bitonic counting network. Each shaded box is a balancer, which switches input requests between its two outputs.

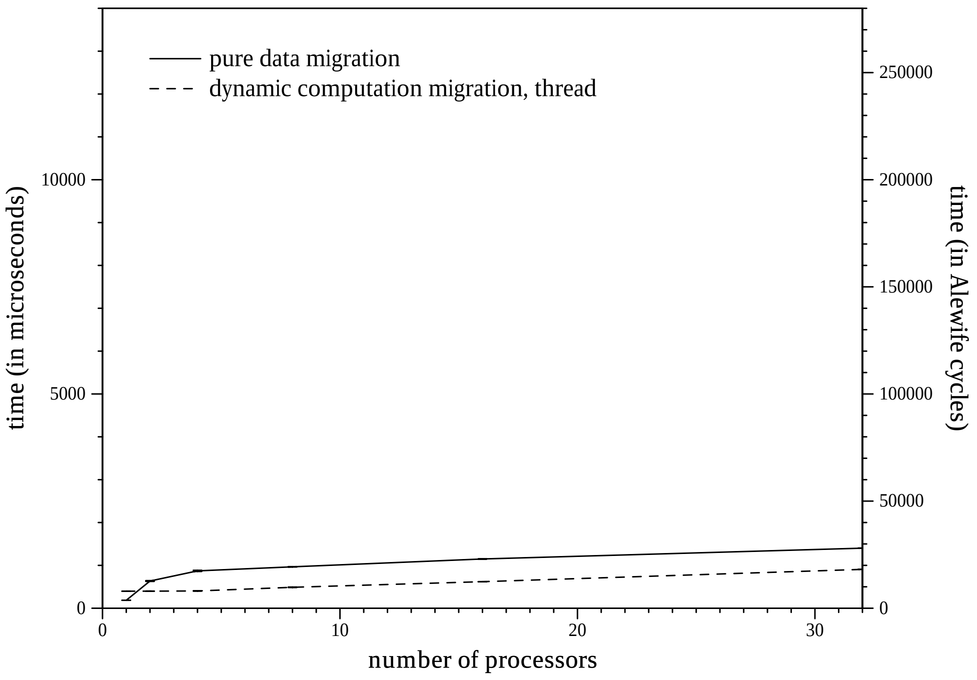

Figure 7-12. Cost per iteration of accessing a counting network on Alewife.

of the counting network is spread across one quarter of the machine. This layout should balance the amount of bandwidth required across any particular communication link.

The experiment creates one thread per processor. Each thread accesses the counting network in a tight loop. On Alewife this experiment was run with up to 32 processors (the largest machine size) accessing the network. On the CM-5 it was run with up to 128 processors. Only the STATIC heuristic was run, because the behavior of the REPEAT heuristic would be virtually identical. At 0% reads, the behavior of the heuristics is to migrate all writes; the extra cost of making decisions in the REPEAT heuristic is small.

The results for the counting network demonstrate that dynamic computation migration can outperform data migration for write-shared data. In particular, on Alewife dynamic computation migration outperforms pure data migration when the number of processors that access a counting network is greater than one. The results also demonstrate that architectural artifacts of the CM-5 can dramatically reduce the performance of dynamic computation migration.

Alewife

Figure 7-12 illustrates the results of using my dynamic computation migration protocol on a counting network on Alewife. Unsurprisingly, dynamic computation migration significantly outperforms data migration, since computation migration does not incur any coherence overhead.

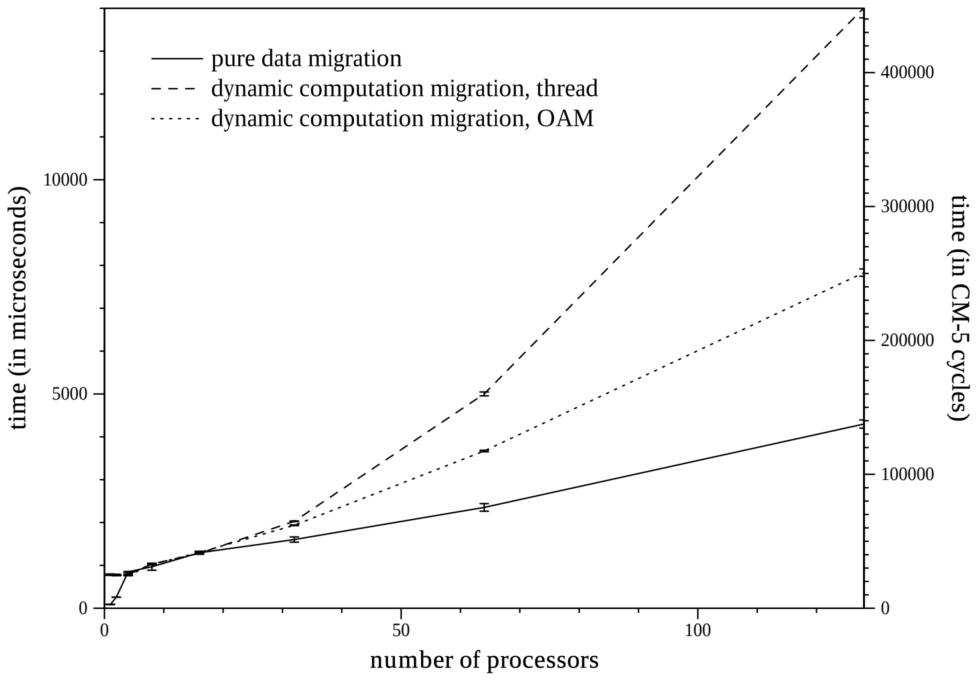

Figure 7-13. Cost per iteration of accessing a counting network on the CM-5.

Despite the small size of the counting network nodes (eight words), the cost of maintaining coherence is still more expensive than moving computation.

Data migration outperforms computation migration when only one processor accesses the counting network. Each balancer that the processor touches will be migrated to that processor. It takes seven iterations for the processor to touch all seventeen balancers that it accesses in the network. Once it has accessed all of those balancers, all subsequent accesses to the network will be local. As a result, data migration with replication is faster than computation migration. However, a counting network would not be used with only one client, since a centralized solution would give better performance.

The cost per iteration for computation migration is the same for one, two, and four clients. The costs are the same because the counting network is four balancers wide. With four or less clients, the clients do not interfere with each other in the counting network. The requests into the counting network will move across the counting network without having to wait. With more than four clients, the counting network begins to behave like a pipeline; with more than twenty-four clients, the pipeline is constantly full.

CM-5

Figure 7-13 illustrates the results of using my dynamic computation migration protocol on a counting network on the CM-5. Several of the results match those on Alewife. For one processor, data migration performs better because the counting network becomes on that processor. The performance of computation migration is the same for one, two, and four processors. For two through sixteen processors, computation migration outperforms data migration. In fact, for sixteen or less processors, the thread version of computation migration actually performs better than the optimistic active message version. This anomaly occurs because the optimistic active message version has enough procedure calls to overflow the register windows. In addition, any optimistic active message that invokes my migration protocol must abort, since my protocol blocks a thread that invokes it. Except for balancers that happen to be accessed locally, these aborts occur on all balancers in the first five stages of the counting network.

For 32 processors and above, data migration performs better than computation migration. The counting network is full, and the cost for computation migration is significantly greater than that of data migration, because of the extra round-trip message that my protocol requires. Not only must the overhead of sending and receiving messages be paid; the extra message also blocks the processor that sends the message. Since a round-trip active message can take as little as 12 microseconds, the sending thread does not yield the processor, which would take about 23 microseconds. As a result, the round-trip message blocks the processor for at least 12 microseconds.

Finally, with 32 or more processors the thread implementation of computation migration is much more expensive than the optimistic active message implementation. The difference in cost occurs because migration acknowledgments are delayed in the thread implementation. In the thread implementation, when a migration request is received, a thread is created; a migration acknowledgment is sent only after the newly created thread is scheduled. In the OAM implementation, on the other hand, the OAM sends a migration acknowledgment immediately (as long as the optimistic active message does not abort). As a result, in the thread implementation the counting network "backs up" quickly, because a number of context switches must occur before a migration acknowledgment is sent.

Discussion

The results for the counting network confirm those in our conference paper [50]; that is, using computation migration improves performance for a counting network. The reason is that a counting network is always written, and writes are expensive when using data migration.

As can be seen by comparing Figures 7-12 and 7-13, the CM-5 is a much less scalable architecture for a counting network. The primary reason is the CM-5's short message size: the extra message that computation migration requires slows down my protocol. Since the sending processor on the CM-5 does not block, this cost (a minimum of 12 microseconds) is not hidden at all. As a result, the cost for computation migration is much greater when using my protocol.

In contrast, the Alewife protocol does not saturate up to 32 processors. In order to further test the scalability of Alewife, I ran the same set of experiments on a four-input, four-output counting network. (Such a network is one-fourth the size of the network shown in Figure 7-11, and is the same as the upper left or lower left subnetworks of the larger network.) Computation migration performs well on the smaller network; dynamic computation migration outperforms pure data migration for two to thirty-two processors.

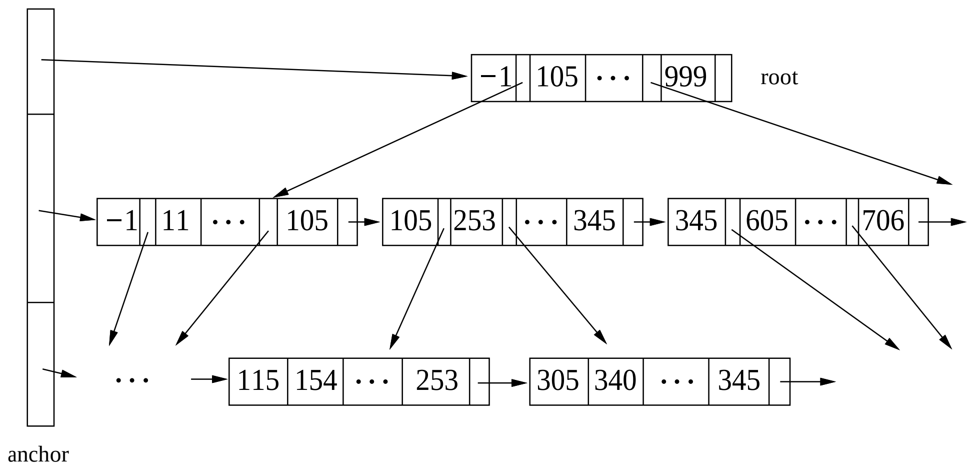

Figure 7-14. A vertical cross-section of a B-tree. This figure diagrams a vertical slice through a three-level B-tree. The anchor is used to locate the root of a B-tree.

7.3.2 B-tree

A B-tree [7] is a data structure designed to represent a *dictionary*, which is a dynamic set that supports the operations *insert*, *delete*, and *lookup*. The basic structure and implementation of a B-tree is similar to that of a balanced binary tree. However, unlike a binary tree, the allowed maximum number of children for each B-tree node is not constrained to two; it can be much larger. Comer [27] presents a full discussion of the B-tree and its common variants.

Bayer and McCreight's original B-tree [7] was designed to support sequential applications. Since then, researchers have proposed many different algorithms to support concurrent operations on the B-tree (e.g., [8, 26, 53, 62, 63, 74, 82, 99]) and have conducted many studies examining their performance (e.g., [29, 54, 62, 99]). Some algorithms [26, 53, 99] are designed to run on a distributed-memory multiprocessor, where the B-tree nodes are laid out across several processors.

Two primary factors limit performance in distributed B-trees: *data* and *resource contention*. Data contention occurs when concurrent accesses to the same B-tree node must be synchronized. The most critical example of data contention is the *root bottleneck*, where an update to the tree's root node causes all incoming B-tree operations to block. Even if no synchronization among concurrent accesses is necessary, performance can still be degraded by resource contention. For example, since every B-tree operation accesses the root node, we would expect the memory module that contains it to be heavily utilized. If the rate of incoming operations exceeds the rate at which requests are serviced, then a bottleneck still exists.

As in the Prelude experiments described in Chapter 5, the B-tree implementation is a simplified version of one of the algorithms proposed by Wang [99] (it does not support the *delete* operation). The form of these B-trees is actually a variant of the "classic" B-tree: keys are stored only in the leaves, and all nodes and leaves contain pointers to the node to their right. As analytically shown by Yao [101], B-tree nodes (in trees with only inserts) are on average approximately 69% full. The values of the keys range from 0 to 1,000,000.

Figure 7-14 includes a diagram of part of a B-tree. The *anchor* is a data structure that is primarily read-only; it is used to locate the *root* of the B-tree. The anchor is only written when the B-tree gains a level. In the experiments the B-tree does not gain levels, which is typical of large B-trees. *Lookup* and *insert* operations traverse the B-tree downwards and rightwards from the root until they find the appropriate leaf. If an insert causes a leaf to overflow, the leaf is split in two; a background thread is started to insert a pointer to the new leaf node into its parent. This splitting process is continued upwards as necessary. If the root winds up being split, a new root is created.

Experiments

The experiments described in this section measure the time for thirty-two processors to access a B-tree. In the experiments an initial B-tree is built with 200,000 keys. Each node or leaf can contain 500 children or keys, respectively; they are approximately 4000 and 2000 bytes in size, respectively. With these parameters, the B-tree is three levels deep. (The B-trees in these experiments are over an order of magnitude larger than those in our Prelude experiments, which is due to the fact that the sizes of the Prelude B-trees were limited by the memory available to the Proteus simulator.) The nodes and leaves of the tree are created randomly across thirty-two processors.

After the B-tree is created, all of the B-tree nodes are then flushed from every processor, so that only the home nodes have valid data. Thirty-two threads are created, one on each processor, each of which repeatedly initiates either lookup or insert requests into the B-tree. This represents a cold access time for the B-tree. After every processor completes a series of cold accesses, the processors repeat the experiment to obtain warm access times for the B-tree. The costs are measured by taking the regression of the average latency of at least 6 runs, where the ith ran for $100i$ iterations.

The B-tree code had to be modified in order to make the two heuristics useful. My heuristics for dynamic computation migration use the distinction between read and write acquisition of a region to decide what to do. On an insert in the original B-tree code, a thread acquires nodes in read-mode until it reaches the bottom level of the B-tree. It then releases the leaf node, reacquires it in write-mode, and acquires nodes in write-mode going across the bottom until finds the correct node. Since an insert always acquires the first leaf node in read-mode before acquiring it in write-mode, the heuristic for writes is effectively ignored when the leaf is reached. In order to make my heuristics effective, the code was changed to always acquire leaf nodes in write-mode. This change does not significantly affect the performance of the B-tree, as fewer than 1% of all inserts need to move across the bottom of the B-tree.

Any background threads (due to splits, which were described above) that are created are started on the home processor of the node that must be written. This optimization is performed in all of my experiments. Writes only use computation migration on the leaves, since any writes that propagate upwards are executed in background threads.

The results demonstrate that dynamic computation migration performs well on a B-tree. The coherence overhead for B-tree nodes is high, particularly because they are large objects.

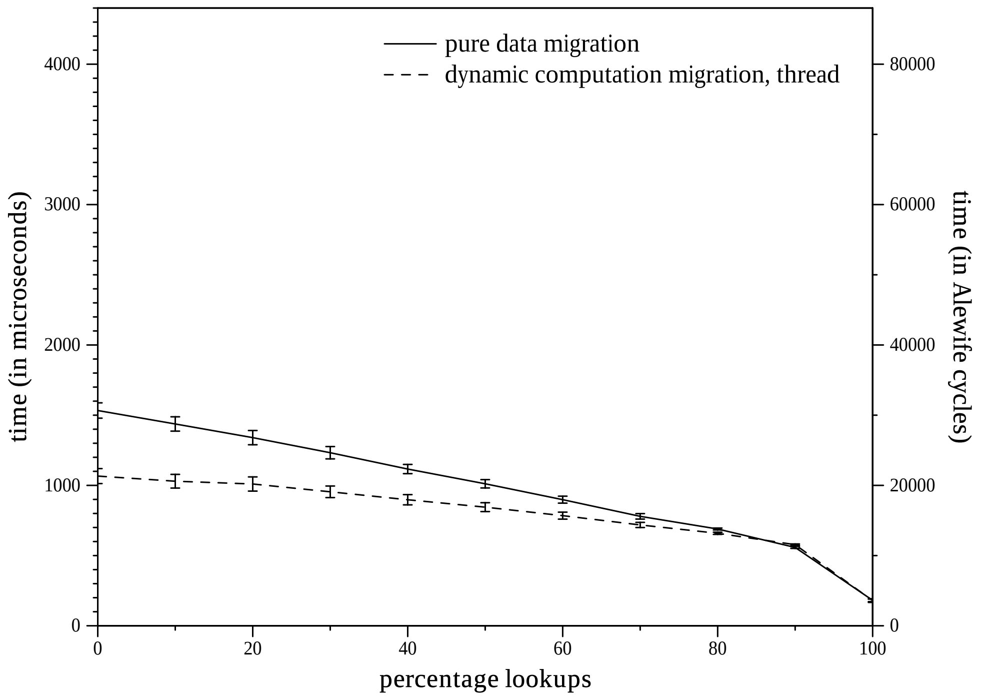

Figure 7-15. Cost per iteration of accessing a B-tree on Alewife using the STATIC heuristic.

Alewife

Figures 7-15 and 7-16 illustrate the results of using my heuristics for a B-tree on Alewife. Paradoxically, the STATIC heuristic generally performs better than the REPEAT heuristic, even when the percentage of lookups is low (and the percentage of writes is high). The reason for this apparent anomaly is that a traversal of the B-tree must visit at least two interior nodes and one leaf node. Visits to the non-leaf nodes from inserts and lookups are always reads, so the rate of writes is very low (writes occur when an insert splits a leaf node). As a result, it is more efficient to always use data migration for the interior nodes, which the STATIC heuristic does. The REPEAT heuristic performs worse because it uses computation migration for some reads of interior nodes.

Although the STATIC heuristic performs better than the REPEAT heuristic when the percentage of lookups is less than 80%, the REPEAT heuristic performs better relative to data migration for lookup percentages between 80% and 90%. For example, at 80% lookups, the REPEAT heuristic performs 23% better than pure data migration, whereas the STATIC heuristic performs only 5% better than pure data migration. The reason is that when the lookup rate increases to 80%, the rate of writes on the second level of the tree drops, so that the effect described in the previous paragraph becomes less significant. Instead, the performance gain for using the REPEAT heuristic on the leaf nodes becomes more significant, which results in better performance.

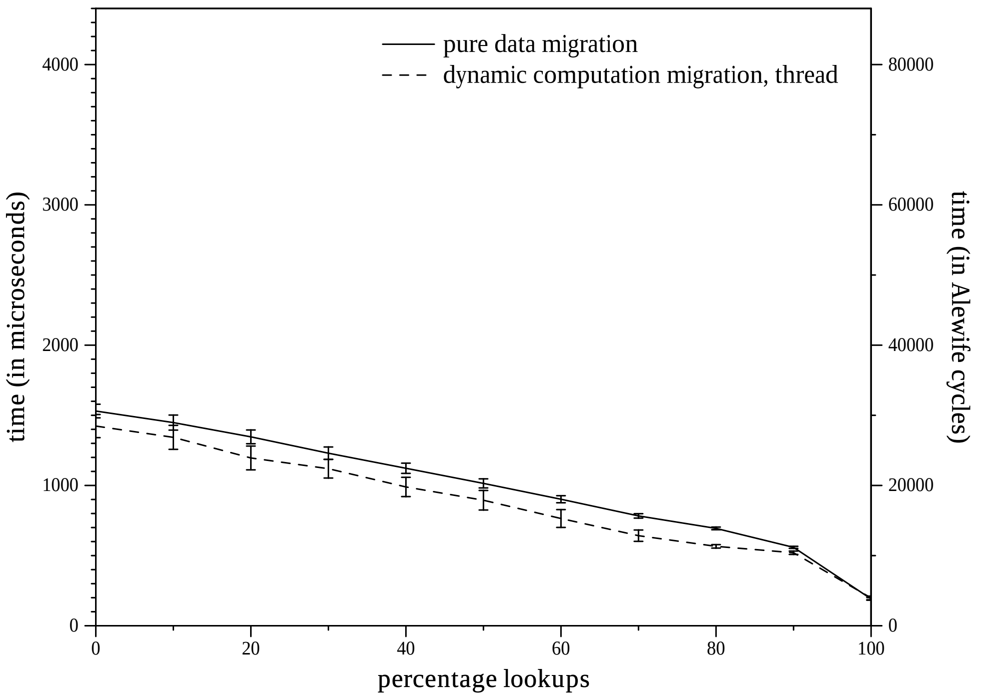

Figure 7-16. Cost per iteration of accessing a B-tree on Alewife using the REPEAT heuristic.

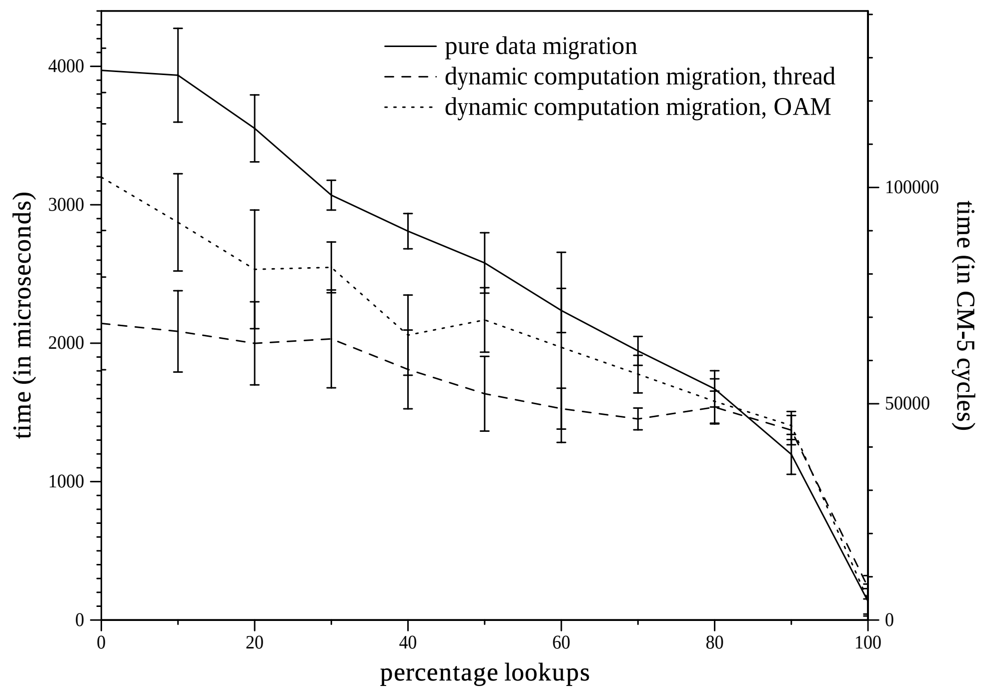

Figure 7-17. Cost per iteration of accessing a B-tree on the CM-5 using the STATIC heuristic.

Although the REPEAT heuristic generally performs worse than the STATIC heuristic, both heuristics outperform pure data migration when the percentage of lookups is lower than 90%. The nodes are large enough to make writes expensive under data migration; as a result, computation migration improves performance when there is a significant proportion of writes.

CM-5

Figures 7-17 and 7-18 illustrate the behavior of my heuristics for a B-tree on the CM-5. Both the STATIC and the REPEAT heuristic perform approximately twice as well as data migration. The reason is that data migration for large objects is expensive on the CM-5; it takes approximately 1/3 of a second to move a 2000-byte object from one processor to another.

The thread implementation performs better than the OAM implementation under the STATIC heuristic, which is surprising. The confidence intervals for measurements of the two implementations overlap on much of the graph, so this is not a conclusive result, but it is likely to be true. This anomaly is most likely due to read contention for invalidated nodes on the second level of the tree. The STATIC heuristic uses data migration for these nodes; as data is being transferred into the network, the network is polled to prevent deadlock. As a result, a read can interrupt another read, which lowers the overall throughput of operations.

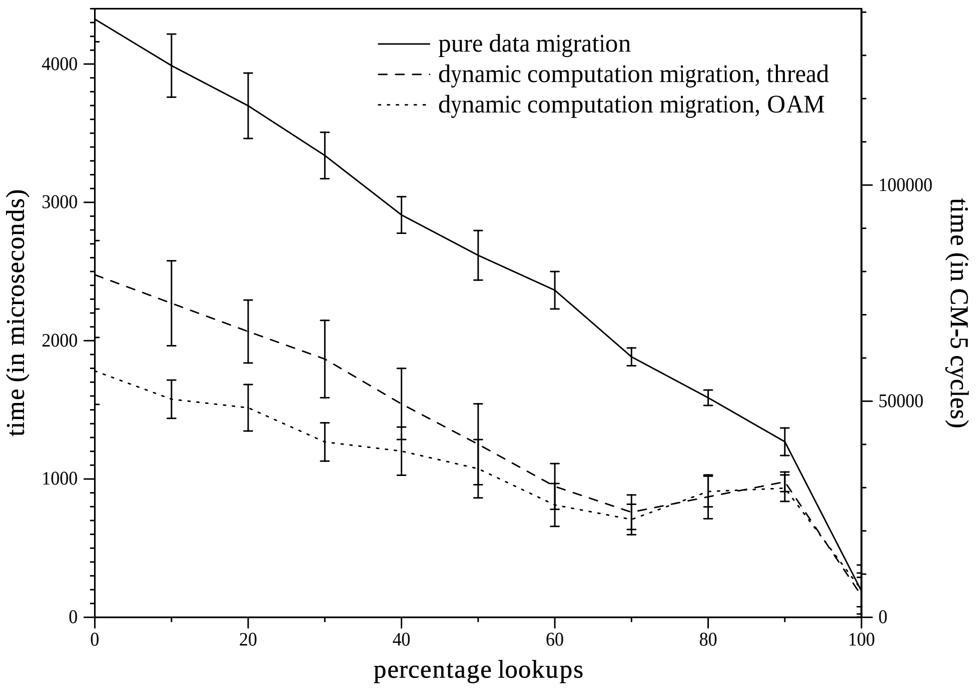

Figure 7-18. Cost per iteration of accessing a B-tree on the CM-5 using the REPEAT heuristic.

The REPEAT heuristic performs at least as well as the STATIC heuristic for the data shown. Due to the high cost of migrating B-tree nodes on the CM-5, the effects that are seen on the second level of the tree on Alewife do not occur on the CM-5. Instead, a reverse effect occurs: the REPEAT heuristic slopes upward from 70% lookups to 90% lookups. This effect occurs because the REPEAT heuristic starts to choose data migration more frequently on the second level of the tree, which is the wrong choice due to the large size of interior nodes (approximately 4Kbytes). Finally, the OAM implementation outperforms the thread implementation, as expected.

Discussion

Again, as in the counting network, the CM-5 performs worse than Alewife. Data migration is much more expensive on the CM-5 because the processor must move data from the network interface to memory. On Alewife the DMA mechanism can be used to move data without the processor's intervention.

Because of the slowness of data migration on the CM-5, the behavior of the two heuristics is reversed relative to Alewife. On Alewife the STATIC heuristic performs better than the REPEAT heuristic because it does less computation migration. Data migration is fairly efficient on Alewife, which makes it more beneficial to allow more caching to occur.

On the CM-5, data migration is very inefficient for large objects. As a result, the better tradeoff is to perform slightly more computation migration; the decreased cache hit rate is offset by the savings in not moving data between processors.

7.4 Summary

This chapter has demonstrated several results. These results fall into three categories. First, they demonstrate the benefits of computation migration, and the effectiveness of the two heuristics for dynamic migration. Second, they illustrate how architectural characteristics affect data migration and computation migration. Third, they confirm our earlier results on optimistic active messages.

Regarding computation migration, the following conclusions can be drawn from my results:

- Using computation migration for writes improves performance, because it reduces the cost of maintaining coherence. On the microbenchmark, the STATIC heuristic outperforms pure data migration for most read/write ratios on both machines; only when there are mostly reads does the STATIC heuristic perform slightly worse. This loss in performance occurs because the decision to use data migration in the STATIC heuristic is made dynamically, not statically.

  On the counting network and the B-tree, the STATIC heuristic almost always outperforms pure data migration on Alewife. Due to various architectural artifacts, computation migration does not perform that well on the counting network on the CM-5.

- Using computation migration for reads can improve performance. On the microbenchmark, the REPEAT heuristic outperforms the STATIC heuristic for most read/write ratios. On the B-tree, the REPEAT heuristic sometimes outperforms the STATIC heuristic on Alewife; it generally outperforms the STATIC heuristic on CM-5. The difference is due to the high cost of data migration on the CM-5.

In addition, several major architectural differences between Alewife and the CM-5 can be seen in my results. These differences have a strong effect on my measurements. First, the combination of slow context switching on the CM-5, combined with the fact that there is no polling during a context switch, degrades performance severely. An active message that arrives during a context switch will be delayed an average of 12 microseconds. This cost hurts the performance of computation migration, because more threads are created when using computation migration.

Second, the limit on message size on the CM-5 degrades performance severely. Any message transfer that sends more than four words requires an extra round-trip active message, which at minimum takes 12 microseconds. This cost hurts the performance of the dynamic migration protocol, which for the measurements described in this chapter always requires messages longer than four words.

Third, the DMA mechanism on Alewife is effective at transferring data between processors. The major reason that DMA is faster than the `scopy` mechanism on the CM-5 is that the processor does not have to move the data itself. The result is that data migration of large objects is significantly cheaper on Alewife, which lowers the relative benefit of computation migration.

Finally, the use of optimistic active messages is in general effective on the CM-5, which confirms our prior results. Using an optimistic active message is more efficient than starting a thread, primarily because the cost of context switching is so high on the CM-5. However, the use of optimistic active messages is ineffective on Alewife, given the various restrictions on active messages in the current version of the kernel and CMMU.

Chapter 8

Related Work

The research described in this dissertation can be viewed as an abstraction of the general parallel programming problem of algorithm/program optimization. When optimizing any particular parallel algorithm, the issue of where computation and data should occur always arises. Typically, most systems have fixed either the data or the computation, and only viewed the movement of computation as a load-balance issue. For example, in high-performance Fortran systems, a useful simplification has been to use the "owner-computes" rule [45, 46, 103]; in these systems the programmer lays out the data and computation always occurs at the data. Conversely, in shared-memory systems, the programmer lays out the computation, and data is pulled to computation when necessary. This dissertation attempts to bridge this gap by exploring the tradeoffs involved when both data and computation can move.

Section 8.1 describes some of the many software DSM systems that have been built or designed, almost all of which have focused on data migration with or without replication. Section 8.2 describes some of the related work on computation migration for multiprocessors. Finally, Section 8.3 describes some of the related work in distributed systems.

8.1 Software Distributed Shared Memory

The combination of a global address space with MIMD multiprocessing and distributed memory is called *distributed shared memory* (or DSM for short). The origins of this term are somewhat vague, but "distributed shared memory" appears to be a transmogrification of the term "shared virtual memory" used by Li and Hudak [67]. A distributed shared memory system provides the programmer with a single address space, even though memory is physically distributed with the processors. Some researchers have used the term "NUMA" (non-uniform memory access) to describe such systems.

Table 8.1 gives a characterization of some of the DSM systems that have been implemented. All of the systems listed provide a global address space. However, such an address space can be implemented at a number of different levels in a system. For example, it can be implemented in the compiler or a library, in the operating system (typically in the virtual memory system), or in the

Implementation	Memory model	Coherence	Example
compiler or library	object	object	Emerald [55], Amber [24], Prelude [100], Orca [6], Midway [102], CRL [51]
operating system	flat	page	Ivy [67], Munin [9], TreadMarks [59]
hardware	flat	cache line	NYU Ultracomputer [38], Cedar [37], IBM RP3[78], DASH [66], Alewife [1]

Table 8.1. Characterization of DSM systems. "Implementation" refers to the level of the system at which shared memory is implemented. "Memory model" describes the organization of the global address space. "Coherence" indicates the unit of coherence. "Example" lists a few example systems; the list is not intended to be exhaustive.

machine itself (typically in the cache system). The software-based systems are described in more detail below.

The address space that the programmer sees depends on the level in the system at which implementation occurs. In compiler and library-level implementations, the programmer sees a global address space in terms of a space of objects. In implementations at the operating system level (where the term "DSM" is widely used) or the hardware level (where the less descriptive term "shared-memory" is typically used), the programmer sees a flat uniform address space, in which pointers can reference memory on any node.

This section overviews the related work in software distributed shared memory systems. Such systems are the implementations that are above machine-level; that is, compiler-level, library-level, or operating-system level implementations as described in Table 8.1 on page 102.

8.1.1 Language-level DSM

Many shared-address space languages provide a global namespace in terms of objects. The roster of parallel object-oriented languages includes Amber [24], Emerald [55], Orca [6], Prelude [100], COOL [23], ABCL/f [91], Concert [57], CST [48], Jade [80], Mentat [39], and Concurrent Aggregates [25]. Although all of these projects are interesting, only some of those relevant to computation migration are discussed in this section. Only one other project, Olden, has investigated computation migration; that project is discussed in more detail in Section 8.2.

This dissertation began with our exploration of static computation migration [50] in the context of Prelude. In that work we demonstrated that simple annotations could be used to specify static computation migration. We also demonstrated the potential benefits of static computation migration for counting networks and B-trees, although for the latter a dynamic choice between data migration and computation migration was clearly necessary.

The Emerald system provides support for the migration of objects in a distributed system. The default mechanism for remote access is remote procedure call, but Emerald provides several options for the arguments to a call. These options can cause an argument object to be "pulled" to an invocation; these mechanisms allow for data migration without replication. An object so invoked could either be pulled for the duration of the call ("call-by-visit") or permanently ("call-by-move"), until the next migration. Amber, which is based on C++, provides similar mechanisms. Neither of these projects explored the use of replication, which is important for improving the performance of read operations.

The COOL language is also based on C++. Similar to Emerald and Amber, it does not support replication, and remote procedure call is the mechanism for remote access that the system provides. COOL provides some mechanisms for data migration, but concentrates on providing mechanisms to control where methods execute. In contrast to Emerald and Amber, however, COOL's mechanisms for managing locality are method-based rather than invocation-based. COOL provides executable "annotations" that describe where and when a method should execute. "Object affinity" for a method results in remote procedure call, whereas "task affinity" is used to execute tasks back-to-back to increase cache reuse.

The SAM runtime system [84] provides a global address space and support for object-oriented parallel programming languages. It provides two data types: single-assignment values and accumulators. Accumulators are managed in a manner similar to migratory data in caching schemes: they are moved from processor to processor as they are accessed.

Most of these projects have involved research on scheduling and replication. For example, the Orca [5] system does dynamic data replication based on compile-time information and runtime access patterns. However, except for our initial work in Prelude, they do not investigate the dynamic movement of computation, except at the granularity of entire threads. Some of these parallel languages do allow the explicit encoding of computation migration. However, making the structure of a program match the layout of its computation is a poor design choice.

Many "shared-memory" C projects distinguish between local and global pointers. Global pointers effectively form a separate address space of objects. For example, CRL [51], on which MCRL is based, provides a separate global namespace of region identifiers. CRL provides an interface that is similar to that of Shared Regions [83]. Another similar project is Cid [76], which also extends C to provide an additional global name space. The Cid project is more ambitious than CRL, as it also provides a multithreading and synchronization model. The Split-C language [31] is another recent language that extends C with a global address space. It differs from other languages in that arithmetic can be performed on global pointers.

The Midway system [11] supports a global address space by requiring that the programmer correctly associate synchronization with every object. Although it provides a single flat address space, the requirement that objects be correctly synchronized by the programmer makes the address space look more like an object-based system. Midway's coherence protocol uses an interesting combination of fixed and variable data sizes for maintaining coherence [102]. The address space is divided into large fixed-size regions, within each of which coherence is maintained on fixed line sizes. Line sizes in different regions can differ, which provides a means to avoid false sharing.

8.1.2 OS-level DSM

Ivy [67] was the first distributed shared memory system implemented at the operating system level. It implemented shared memory within the virtual memory system: instead of using the disk as backing store, shared pages were paged from remote processors.

The Munin project [9] investigated the use of type-specific coherence protocols. It demonstrated that the choice of the most efficient coherence protocol depends on the access pattern of the type. Munin allowed a programmer to choose among different coherence protocols for different types of data. However, as in other distributed shared memory projects, it did not integrate their coherence protocol with computation migration.

In his dissertation Carter briefly analyzes the advantages of using RPC to access a job queue [18]. Since a job queue is frequently written by many processors, data migration is a poor choice. RPC is preferable to computation migration, since the purpose of using a job queue is to distribute work, and migrating computations to a job queue would do the opposite.

The TreadMarks system [59] is technically not implemented in the operating system, as it is implemented entirely at user-level. However, it makes use of the virtual memory system to maintain coherence, and as a result is closer in spirit to other operating-system based systems.

The Olden project [81] is another distributed shared memory that uses the virtual memory system to detect accesses to non-local pointers. The following section discusses Olden in more depth, as it also provides support for computation migration.

8.2 Computation Migration

Rogers *et al.* [81] have also developed mechanisms for migrating single activation records: on any access to remote data, they always migrate the top activation. The idea of migrating activations is similar to our work, but their approach is more restrictive than ours. Migrating an activation is often, but not always, the best approach. We let the user choose the appropriate mechanism; this flexibility can be vital for achieving good performance.

In more recent work, Rogers and Carlisle [17] have expanded the Olden project to give language support for statically determining when to migrate computation or data. The programmer is required to give the compiler indirect knowledge of how data is laid out by specifying how likely a data structure traversal is to cross processors. Such a strategy works well for scientific applications, where the layout of data structures is fairly predictable. However, it is unlikely to work for truly dynamic data structures such as B-trees.

Fowler and Kontothanassis [35] have suggested performing what they call "temporary thread migration". However, they do not compare and analyze the tradeoffs involved in using computation migration versus data migration.

Load balance can conflict with locality management in parallel systems. However, locality is a generally more important issue than load balance. For instance, Markatos and LeBlanc have shown that for NUMA shared-memory multiprocessors, locality management can have a much larger impact on performance than load balancing [71]. This effect is due to the high cost of communication in distributed-memory systems.

8.3 Distributed Systems

Remote evaluation [88] is a mechanism for distributed systems that would enable an implementation of computation migration. The idea is to have servers export an *eval* call to clients, instead of a fixed RPC interface. The use of this call can reduce communication demand, since clients can create their own procedures to batch multiple calls together.

Service rebalancing [44] is a related mechanism that moves inactive modules between clients and servers in an RPC system. Service rebalancing allows client modules to become server modules, and vice versa; its use is intended to support load balancing.

The Thor object-oriented database [68] provides support for batching cross-domain calls to reduce communication [104]. Unlike remote evaluation and service rebalancing, Thor does not allow clients to send arbitrary requests to a server. Instead, a client can combine the methods in a server's interface using simple control structures.

All of the above systems concentrate on reducing message traffic for RPC systems. None of them integrate their mechanisms with data migration. In addition, protection between a client and server is important in their systems; in my research it is not an issue.

Research of a similar flavor also occurs in query optimization for distributed databases [33]. The primary distinction between databases and distributed shared memory systems is that the amount of data being accessed is much larger; instead of small region-based accesses, accesses take the form of queries over relations or whole databases. The result is that query optimization concentrates on minimizing the volume of data that must be moved; techniques such as semijoin [10] are used to avoid moving entire relations. The other difference is that such techniques typically concentrate on the execution of a single query, whereas my techniques look at the aggregate performance of multiple accesses from different processors.

Chapter 9

Conclusions

This chapter summarizes the contributions and results of this dissertation. It then discusses some possible extensions of my work, outlines directions for future research, and concludes with some observations about parallel computing.

9.1 Summary

This dissertation has described a new mechanism for remote access, computation migration. It has discussed design issues for computation migration, and has described two DSM systems that support computation migration. It has also evaluated two heuristics for dynamically choosing between computation migration and data migration. The following conclusions can be drawn from the performance results described in this dissertation:

- Computation migration is a useful mechanism that complements RPC and data migration with replication. Computation migration performs well with respect to data migration when maintaining coherence is expensive: that is, when the data is large, or when there is a high percentage of write accesses to the data. Computation migration is particularly useful in applications that use graph traversal, because it allows computation to "crawl" through a distributed data structure.

- For the application kernels measured, write accesses should use computation migration. Our Prelude results show that for a counting network, a software implementation of computation migration can outperform a hardware implementation of data migration. In general, synchronization objects will tend to be written frequently, which makes them good candidates for computation migration. The MCRL results demonstrate that a dynamic choice to use computation migration for writes performs well, even with the overhead of making a dynamic choice.

- Using data migration for reads can suffice if data is either strongly read-shared or strongly write-shared. The STATIC heuristic always uses computation migration for writes and data migration for reads. It performs well for a B-tree, where the upper levels of the tree are strongly read-shared; using computation migration for writes improves performance by 44%, relative to pure data migration, when the operation mix consists only of inserts. However, as demonstrated by a simple microbenchmark, the STATIC heuristic does not perform as well relative to pure computation migration when there is a near-equal mix of reads and writes.

- Although the STATIC heuristic performs well when there are many reads or many writes, the REPEAT heuristic performs better when there is a near-equal mix of reads and writes. Microbenchmark measurements indicate that the REPEAT heuristic correctly chooses to use computation migration when the numbers of reads and writes are approximately equal. The REPEAT heuristic can perform better than the STATIC heuristic when the cost to move data is high. For a B-tree on Alewife with 80% lookups and 20% inserts, using a dynamic choice for reads improves performance 23% relative to pure data migration; always using data migration for reads only improves performance by 5%.

In addition, several conclusions can be drawn about the two machine architectures on which MCRL has been implemented. With respect to the differences between Alewife and the CM-5, the following results have been demonstrated. First, the SPARC register windows hurt performance on the CM-5. It is not clear that register windows are useful on sequential machines; Wall has demonstrated that link-time register allocation can perform nearly as well [96]. However, the presence of register windows decreases the performance of the CM-5. The register windows raise the cost of context switching on the CM-5. In addition, they raise the cost for deeply nested procedure calls; when running with active messages the call stack is deeper than in sequential programs. Second, the DMA mechanism on Alewife is effective in moving data. DMA is particularly effective in MCRL, because extra data moves to set up DMA are not required: a region's data acts as a message buffer.

Finally, with respect to optimistic active messages, the following conclusions can be drawn. First, the use of optimistic active messages is in general effective on the CM-5, which confirms our prior results. Using an optimistic active message is more efficient than starting a thread, primarily because the cost of context switching is so high on the CM-5. Second, the use of optimistic active messages is ineffective on Alewife, given the restrictions on active messages in the current version of the kernel and CMMU. A faster implementation might be possible using polling on Alewife. However, that is not the natural programming style, given that interrupts are fast.

9.2 Future Trends

As networks continue to evolve, we see two trends that are unlikely to change:

- The communication bandwidth available in parallel systems is increasing. The raw bandwidth provided by networks is increasing. In addition, the ability to access network interfaces from user space is also reducing the software bottleneck on communication bandwidth.

- While network latencies are decreasing, processor cycle times are decreasing even faster. As a result, network latencies (in terms of processor cycles) are increasing, and the cost for transmitting messages will remain significant compared to the cost of local computation.

The first trend means that the size of messages will matter less. As a result, the cost to send "large" data will decrease, and the effects of data size will be less when using data migration. On the other hand, the marginal cost to migrate additional computation state will decrease as well, which will decrease the cost to migrate computation with a large amount of state.

Although the size of data will matter less when using data migration, computation migration will still be effective in reducing coherence traffic. The second trend means that the cost of communication will matter more. As a result, mechanisms that reduce the number of messages in a distributed shared memory system will become more important, and computation migration will remain useful.

9.3 Extensions

It would be interesting to remeasure my CM-5 experiments on a CM-5E. The CM-5E has support for longer messages than the CM-5; messages can contain up to 18 words. With longer messages, the extra round-trip message that my protocol requires on the CM-5 would not be necessary for the measurements described in this dissertation. As a result, the performance results would be more interesting, and would allow for a better analysis of dynamic computation migration on the CM-5 architecture.

A useful extension to MCRL would be the provision for two types of read-mode. As described in Section 7.3.2, the B-tree code had to be rewritten for my heuristics, because the decisions they make are based on whether operations are reads or writes. Instead of changing read-mode acquisitions to write-mode, which limits concurrency, it would be cleaner to provide a hybrid read / write mode. Such a mode would tell the runtime system that a thread is likely to upgrade its copy to write-mode.

Another useful extension would be to investigate a protocol that allowed a client to dynamically choose between data and computation migration. The STATIC heuristic could be used with such a protocol, although the REPEAT heuristic could not. In addition, although the simple heuristics that have been described are effective, there is a large design space of heuristics that can be explored. For example, it may be possible to improve them by taking load into account.

It would also be worth experimenting with the migration of partial or multiple activation records. Although some of the relevant issues have been discussed, neither migration of partial records nor migration of multiple records has been implemented. It would be useful to investigate mechanisms for migrating computation at different granularities, but it is not clear what applications could profitably make use of such mechanisms.

Finally, it would be useful to analyze the number of coherence messages that computation migration saves over data migration. Measured message counts would be useful, as well as an analytic model of message patterns. In addition, a comparison of computation migration to different coherence protocols would be useful, since the comparison using MCRL only compares computation migration to a single coherence protocol.

9.4 Future Work

My intention has been to treat MCRL as a compiler target. Instead of building a parallel C compiler with support for computation migration, a more interesting research direction would be to build a compiler for an object-oriented programming language that translates to MCRL. There would be many open issues to deal with in such a compiler, such as how to hoist `rgn_map`, `rgn_start_read`, and `rgn_start_write` out of blocks of code (as discussed in Section 6.2.1).

Another direction for future work is the investigation of automatic mechanisms to make use of dynamic computation migration. Although it is not difficult to insert program annotations for migration, it would be preferable to automate the decision about where computation migration should occur. One possibility would be to combine the analyses that the Olden compiler performs with a dynamic computation migration protocol.

The exploration of user annotations to control dynamic migration directly could be fruitful. Such annotations could allow the user to specify hints for dynamic migration in a region-specific manner. For example, in the B-tree it might be possible to use the level of a node when deciding whether to use computation or data migration. Since it is implemented in software at user-level, MCRL is sufficiently flexible to allow for application-level decisions.

When computation migration outperforms data migration, it may also achieve a better cache utilization than data migration. In other words, since data migration replicates data on multiple processors, it has a larger cache footprint than computation migration. It would be valuable to assess whether computation migration does achieve any benefit from this effect.

This dissertation has examined computation migration in homogeneous, tightly-coupled parallel systems. The results should remain valid for DSM systems on more loosely-coupled parallel systems such as workstation clusters, given the convergence of local area networks and supercomputer networks. An open area of investigation will be the use of computation migration in heterogeneous systems, especially because the same program image would not be loaded on every processor. In a heterogeneous system, computation migration may involve sending code, which could be expensive.

Finally, this dissertation has examined the performance impact of computation migration, but in future parallel systems (for example, workstations on a wide-area network) other concerns may be an issue. The ownership of data could well be an important factor in deciding whether to allow data migration. For example, the use of a dynamic protocol to allow a server to choose whether to export data could be mandatory if some servers refuse to export data.

9.5 Observations

This dissertation has described a new mechanism, computation migration, that can reduce the cost of coherence for certain types of data. Other research on reducing coherence costs has evaluated the effect of using different patterns for migrating data. Computation migration takes a complementary approach to reducing coherence costs: instead of migrating data, it migrates computation. Such an approach is an example of how examining problems from different perspectives can lead to interesting new solutions.

Although distributed shared memory systems provide a simple programming model that allows for the exploitation of locality, the choice of implementation varies widely. Computer architects build shared-memory machines; operating systems researchers build virtual-memory based systems; and language designers build parallel object-oriented languages. Unfortunately, there has not been a great deal of convergence among these three research communities, and there also has not been analysis of the tradeoffs among the three different approaches.

High-level object-based parallel programming languages are the easiest platform for providing distributed shared memory. Software implementations have the advantage that the unit of sharing is flexible. Fixed-size units are either too large, which leads to false sharing, or are too small, which leads to more communication. Software implementations also have the advantage that new mechanisms such as computation migration can be added. Hardware is not modifiable, and as such is less flexible. Finally, language-based approaches have the enormous potential advantage of being able to take advantage of high-level program analysis.

Finally, the biggest difficulty with parallel machines is their lack of usability. In particular, there is a dearth of debugging support for parallel systems. Debugging a multithreaded parallel problem is extremely difficult. Not only are parallel programs inherently more difficult to debug than sequential ones; the debugging tools that are provided by current systems are in general worse than the tools on sequential machines. Parallel machines will only take the first step towards being general-purpose when there are good programming tools available for them.

Appendix A

CRL and MCRL Internals

This appendix summarizes the internal structure of CRL and MCRL in terms of the state diagrams that they use. The diagrams for the home state machines are identical for CRL and MCRL.

In the following diagrams, solid arrows represent transitions due to user calls. In the home diagrams, thickly dashed arrows represent transitions due to messages from the remote diagrams, and vice versa. Thinly dashed arrows are similar to the thickly dashed arrows, except that the messages that they represent are in response to user calls. States with “Rip” (Read-in-progress) in their names mean that a read is in progress; states with “Wip” (Write-in-progress) in their names mean that a write is in progress. Remote states with “Req” in their names mean that a request for a read or write copy is outstanding. Home states with “Iip” (Invalidate-in-progress) in their names mean that invalidates are outstanding from the home.

Figure A-1 illustrates the state machine for home nodes in CRL and MCRL; Figure A-2 illustrates the state machine for remote nodes in CRL. Figure A-3 illustrates the new state machine for remote nodes in MCRL. The “RemoteMip” state means that a potential migration is in progress. The meanings of the message labels in these figures are given in Table A.1.

CRL regions are laid out with a header in front of them; this header contains coherence information. The headers have not been optimized for size, and as a result are rather large. The header region for CRL is 22 words on Alewife (2 of the words are to work around a DMA bug), and 26 words on the CM-5 (1 word is to pad out to a double word). In MCRL these values go up to 44 words (1 extra word to pad out to a double word) and 46 words (one word of pad can be removed). The REPEAT heuristic requires 5 more words to store heuristic state; the STATIC heuristic does not require any extra storage.

Call	Meaning
Server messages	
start_read	user calls `rgn_start_read` or `rgn_start_local_read`
start_write	user calls `rgn_start_write` or `rgn_start_local_write`
start_local_read	user calls `rgn_start_local_read`
start_local_write	user calls `rgn_start_local_write`
end_read	user calls `rgn_end_read`
end_readi	user calls `rgn_end_read` and an invalidate has arrived
end_write	user calls `rgn_end_write`

Message	Meaning
Server messages	
e-req	request for a write copy
m-req	request for a write copy when the client has a read copy
m-req0	request for a write copy when the requester and the home have the only read copies
s-req	request for a read copy
flush	client flushes its copy
flush0	client flushes its copy and it had the last read copy other than the home
inv's back	all invalidates have been acknowledged
release	a client downgrades its write copy to a read copy
Client messages	
e-ack	acknowledgment of e-req
m-ack	acknowledgment of an m-req
s-ack	acknowledgment of an s-req
inv	an invalidate has arrived

Table A.1. Calls and message types in CRL and MCRL state diagrams

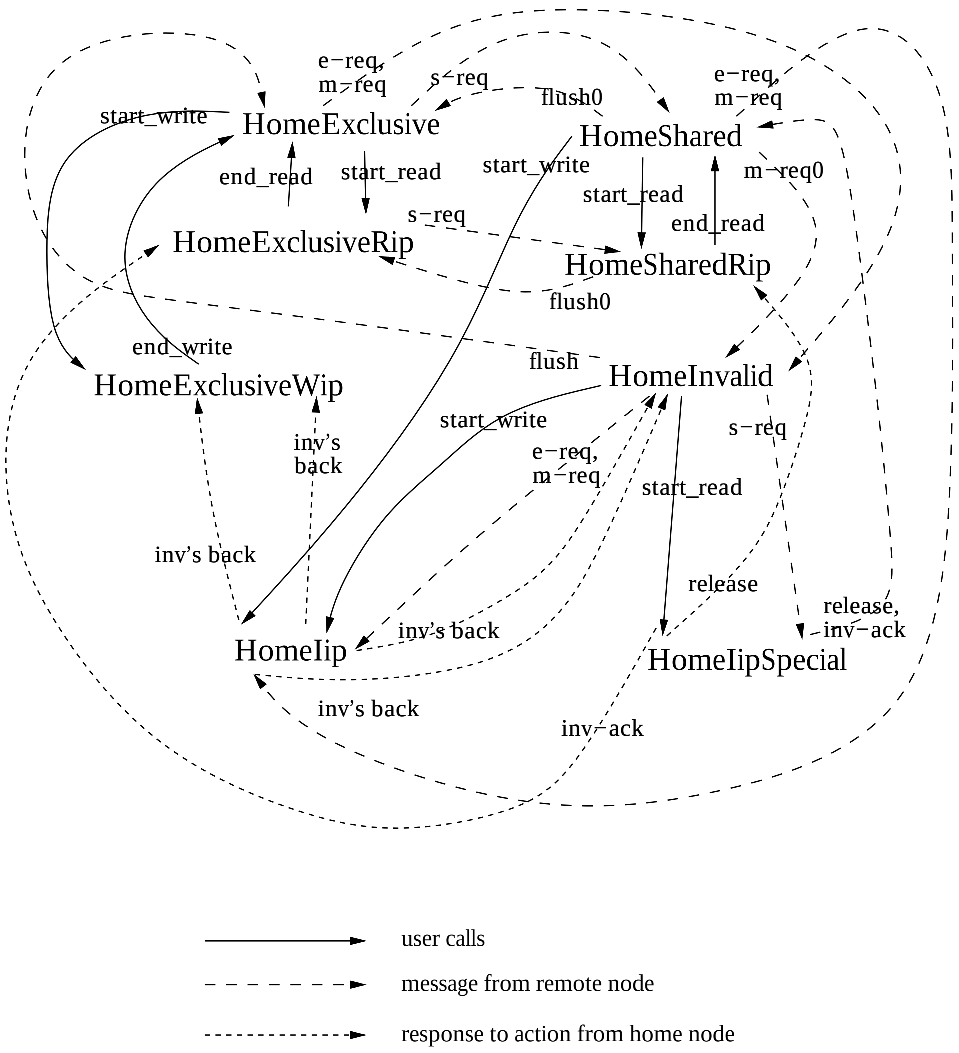

Figure A-1. Home state diagram for CRL and MCRL

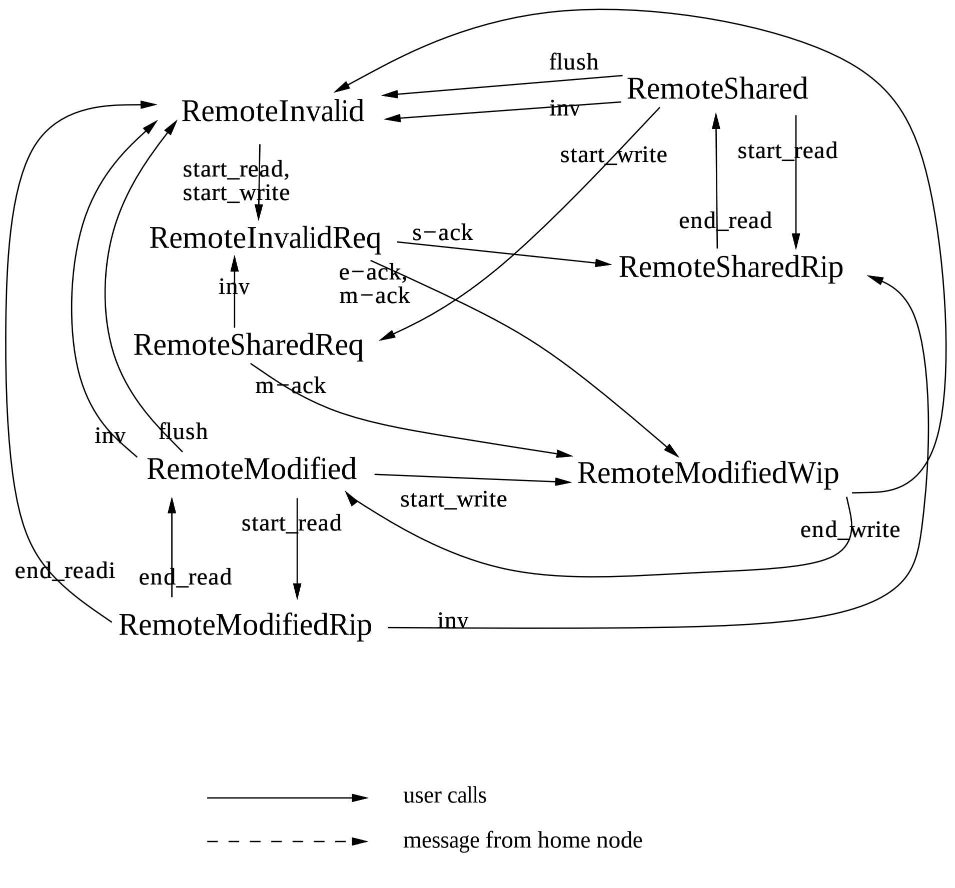

Figure A-2. Remote state diagram for CRL

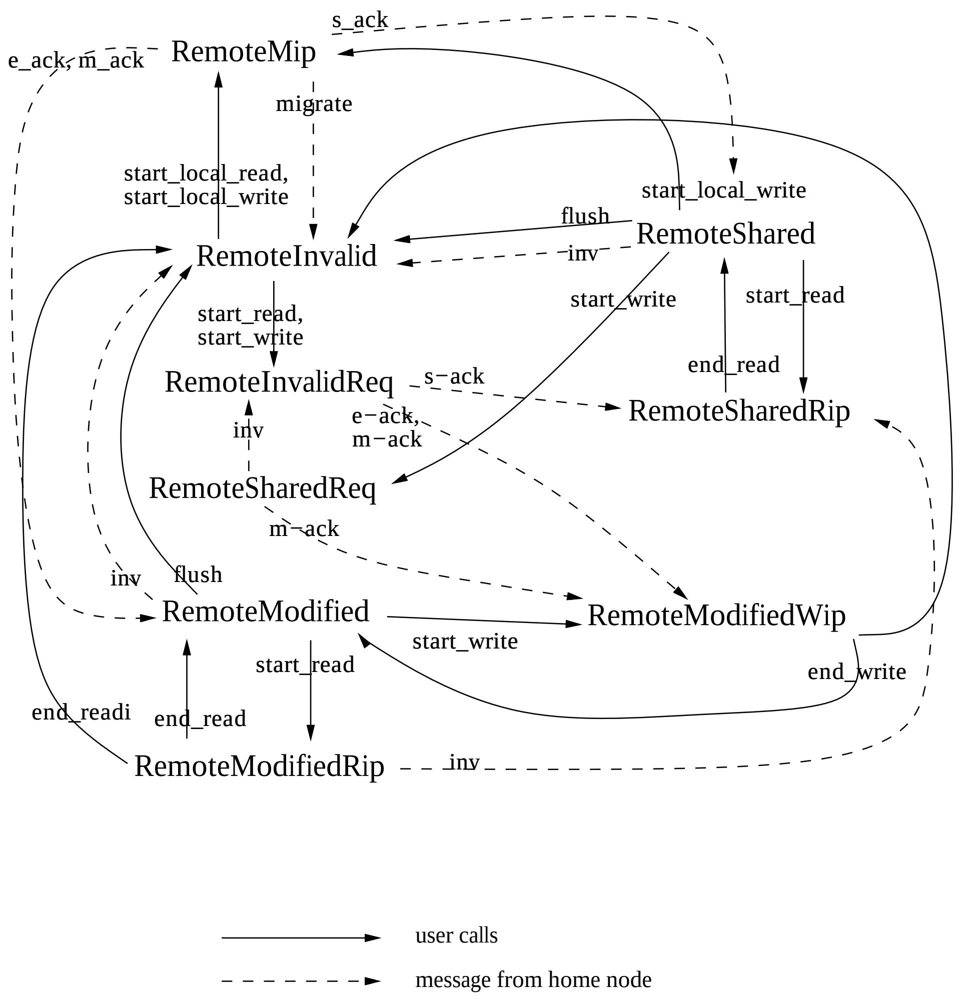

Figure A-3. Remote state diagram for MCRL

Appendix B

LIFO Versus FIFO Scheduling

This appendix describes the difference in measurements that can occur due to LIFO scheduling of threads, as mentioned in Section 6.4.1.

Under LIFO scheduling, the processors that happen to arrive last will have their computation executed first. These processors will be able to request more computation, and will complete earlier than the other processors. Under FIFO scheduling, all processors will finish at approximately the same time. Aggregate throughput does not change, but the average latency that is measured decreases dramatically.

Figure B-1 illustrates a simplified form of this effect. Each number represents a job from a particular processor; there are four processors, and each submits four jobs to a server. The aggregate time to finish all 16 jobs is the same, so the throughput of jobs through the system under either scheduling discipline. However, the average latency to completion for each processor is 50% as long under FIFO scheduling.

Under both scheduling disciplines the aggregate throughput of processor completion is 1 processor per 4 job times. Under LIFO scheduling, the average latency seen by a processor is 10 job times. Under FIFO scheduling, the average latency is 14.5 job times.

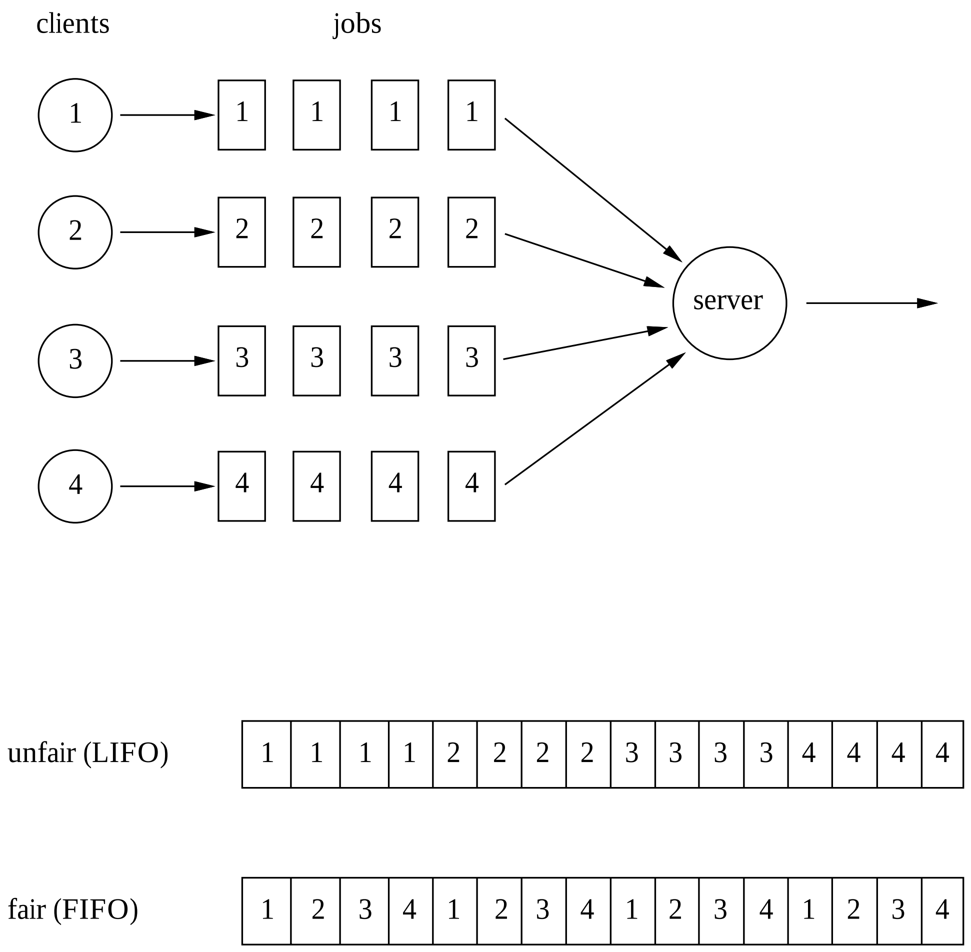

Figure B-1. LIFO versus FIFO scheduling

Appendix C

Regression Formulas

This appendix contains the regression formulas used to compute the performance results in this dissertation. These equations, and tabulated values of Student's *t* distribution, can be found in any standard statistics text [73].

For a set of n points of the form (x_i, y_i), the slope $\hat{\beta}_1$ of the least-squares line is given by Equation C.1.

$$\hat{\beta}_1 = \frac{n \sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{C.1}$$

The 95% confidence interval on the slope is given by

$$\hat{\beta}_1 \pm \frac{t_{.025} s}{\sqrt{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}}$$

where s is the standard deviation of the random error and $t_{.025}$ is based on $(n-2)$ degrees of freedom.

The variance of the random error, s^2, is given by Equation C.2.

$$s^2 = \frac{\sum_{i=1}^{n} y_i^2 - \frac{\left(\sum_{i=1}^{n} y_i\right)^2}{n} - \hat{\beta}_1 \left(\sum_{i=1}^{n} x_i y_i - \frac{\left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n}\right)}{n-2} \tag{C.2}$$

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 22–24, 1995.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[3] T.E. Anderson, E.D. Lazowska, and H.M. Levy. "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors". *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[4] J. Aspnes, M. Herlihy, and N. Shavit. "Counting Networks". *Journal of the ACM*, 41(5):1020–1048, September 1994.

[5] H.E. Bal and M.F. Kaashoek. "Object Distribution in Orca using Compile-Time and Run-Time Techniques". In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '93*, pages 162–177, September 26–October 1, 1993.

[6] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[7] R. Bayer and E.M. McCreight. "Organization and Maintenance of Large Ordered Indexes". *Acta Informatica*, 1(3):173–189, 1972.

[8] R. Bayer and M. Schkolnick. "Concurrency of Operations on B-trees". *Acta Informatica*, 9:1–22, 1977.

[9] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence". In *Proceedings of the 2nd Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[10] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve, and J.B. Rothnie, Jr. "Query Processing in a System for Distributed Databases". *ACM Transactions on Database Systems*, 6(4):602–625, December 1981.

[11] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. "The Midway Distributed Shared Memory System". In *Proceedings of COMPCON Spring 1993*, pages 528–537, San Francisco, CA, February 22–26, 1993.

[12] A.D. Birrell and B.J. Nelson. "Implementing Remote Procedure Calls". *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[13] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. "Cilk: An Efficient Multithreaded Runtime System". In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 19–21, 1995.

[14] E.A. Brewer, F.T. Chong, L.T. Liu, S.D. Sharma, and J. Kubiatowicz. "Remote Queues: Exposing Message Queues for Optimization and Atomicity". In *Proceedings of the 1995 Symposium on Parallel Algorithms and Architectures*, pages 42–53, Santa Barbara, California, July 16–18, 1995.

[15] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. "Proteus: A High-Performance Parallel Architecture Simulator". Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991. A shorter version appears in the *Proceedings of the 1992 Conference on Measurement and Modeling of Computer Systems*.

[16] E.A. Brewer and B.C. Kuszmaul. "How to Get Good Performance from the CM-5 Data Network". In *Proceedings of the 1994 International Parallel Processing Symposium*, pages 858–867, Cancun, Mexico, April 26–29, 1994.

[17] M.C. Carlisle and A. Rogers. "Software Caching and Computation Migration in Olden". In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.

[18] J.B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*. PhD thesis, Rice University, August 1993.

[19] 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, Pacific Grove, CA, October 13–16, 1991.

[20] D. Chaiken, J. Kubiatowicz, and A. Agarwal. "LimitLESS Directories: A Scalable Cache Coherence Scheme". In *Proceedings of the 4th Conference on Architectural Support for Programming Languages and Systems*, pages 224–234, April 1991.

[21] D.L. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 314–324, Chicago, IL, April 1994.

[22] S. Chakrabarti, E. Deprit, E. Im, J. Jones, A. Krishnamurthy, C. Wen, and K. Yelick. "Multipol: A Distributed Data Structure Library". Available at URL http://www.cs.berkeley.edu/projects/parallel/castle/multipol, December 1994.

[23] R. Chandra, A. Gupta, and J.L. Hennessy. "COOL: An Object-Based Language for Parallel Programming". *IEEE Computer*, 27(8):13–26, August 1994.

[24] 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 Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[25] A.A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, MA, 1993.

[26] A. Colbrook, E. Brewer, C. Dellarocas, and W.E. Weihl. "Algorithms for Search Trees on Message-Passing Architectures". In *Proceedings of the 1991 International Conference on Parallel Processing*, volume III, pages 138–141, 1991.

[27] D. Comer. "The Ubiquitous B-Tree". *ACM Computing Surveys*, 11(2):121–128, June 1979.

[28] E.C. Cooper and R.P. Draves. "C Threads". Technical Report CMU-CS-88-154, CMU Computer Science Dept. June 1988.

[29] P.R. Cosway. "Replication Control in Distributed B-Trees". Master's thesis, Massachusetts Institute of Technology, February 1995.

[30] A.L. Cox and R.J. Fowler. "Adaptive Cache Coherency for Detecting Migratory Shared Data". In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 98–108, San Diego, CA, May 16–19, 1993. Stenström et al. [89] have a similar algorithm.

[31] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. "Parallel Programming in Split-C". In *Proceedings of Supercomputing '93*, pages 262–273, Portland, OR, November 1993.

[32] D.L. Eager, E.D. Lazowska, and J. Zahorjan. "The Limited Performance Benefits of Migrating Active Processes for Load Sharing". In *Proceedings of the 1988 Conference on Measurement and Modeling of Computer Systems*, pages 63–72, Santa Fe, NM, May 24–27, 1988.

[33] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1989.

[34] D.R. Engler, G.R. Andrews, and D.K. Lowenthal. "Filaments: Efficient Support for Fine-Grain Parallelism". Technical Report 93-13, University of Arizona, April 1993.

[35] R.J. Fowler and L.I. Kontothanassis. "Improving Processor and Cache Locality in Fine-Grain Parallel Computations using Object-Affinity Scheduling and Continuation Passing

(Revised)”. Technical Report 411, University of Rochester Computer Science Department, June 1992.

[36] V.W. Freeh, D.K. Lowenthal, and G.R. Andrews. “Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations”. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, CA, November 14–17, 1994.

[37] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. “Cedar — a Large Scale Multiprocessor”. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 524–529, August 23–26, 1983.

[38] A. Gottlieb, R. Grishman, C. P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. “The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer”. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[39] A.S. Grimshaw. “Easy-to-Use Object-Oriented Parallel Processing with Mentat”. *IEEE Computer*, 26(5):39–51, May 1993.

[40] A. Gupta and W.D. Weber. “Cache Invalidation Patterns in Shared-Memory Multiprocessors”. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[41] T. Härder. “Observations on Optimistic Concurrency Control Schemes”. *Information Systems*, 9(2):111–120, June 1984.

[42] D.S. Henry and C.F. Joerg. “A Tightly-Coupled Processor-Network Interface”. In *Proceedings of the 5th Conference on Architectural Support for Programming Languages and Systems*, pages 111–122, October 1992.

[43] M. Herlihy, B.H. Lim, and N. Shavit. “Scalable Concurrent Counting”. *ACM Transactions on Computer Systems*, To appear.

[44] E.H. Herrin II and R.A. Finkel. “Service Rebalancing”. Technical Report 235-93, University of Kentucky Department of Computer Science, May 12, 1993.

[45] High Performance Fortran Forum. *High Performance Fortran Language Specification*, version 1.0 edition, May 1993.

[46] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. “An Overview of the Fortran D Programming System”. In *Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[47] W. Horwat. “A Concurrent Smalltalk Compiler for the Message-Driven Processor”. Technical Report 1080, MIT Artificial Intelligence Laboratory, May 1988. Bachelor’s thesis.

[48] W. Horwat, A.A. Chien, and W.J. Dally. “Experience with CST: Programming and Implementation”. In *Proceedings of the SIGPLAN ’89 Conference on Programming Language Design and Implementation*, pages 101–109, Portland, OR, June 21–23 1989.

[49] W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, D.A. Wallach, and W.E. Weihl. “Efficient Implementation of High-Level Languages on User-Level Communication Architectures”. Technical Report MIT/LCS/TR-616, MIT Laboratory for Computer Science, May 1994.

[50] W.C. Hsieh, P. Wang, and W.E. Weihl. “Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems”. In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, pages 239–248, San Diego, CA, May 1993.

[51] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. “CRL: High-Performance All-Software Distributed Shared Memory”. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain, CO, December 3–6, 1995. To appear. http://www.pdos.lcs.mit.edu/crl.

[52] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. “CRL: High-Performance All-Software Distributed Shared Memory”. Technical Report MIT/LCS/TM-517, MIT Laboratory for Computer Science, March 1995. This is an earlier version of [51].

[53] T. Johnson and A. Colbrook. “A Distributed Data-Balanced Dictionary Based on the B-Link Tree”. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 319–324, Beverly Hills, CA, March 23–26, 1992.

[54] T. Johnson and D. Shasha. “A Framework for the Performance Analysis of Concurrent B-tree Algorithms”. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, Nashville, TN, April 2–4, 1990.

[55] E. Jul, H. Levy, N. Hutchison, and A. Black. “Fine-Grained Mobility in the Emerald System”. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[56] G. Kane and J. Heinrich. *MIPS RISC Architecture*. MIPS Computer Systems, Inc., 1992.

[57] V. Karamcheti and A. Chien. “Concert — Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware”. In *Proceedings of Supercomputing ’93*, pages 598–607, Portland, OR, November 15–19, 1993.

[58] A.R. Karlin, K. Li, M.S. Manasse, and S. Owicki. “Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor”. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 41–55, Pacific Grove, CA, October 13–16, 1991.

[59] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. “TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems”. In *Proceedings of 1994 Winter USENIX*, pages 115–131, San Francisco, CA, January 17–21, 1994.

[60] D.A. Kranz, R. Halstead, and E. Mohr. "Mul-T: A High-Performance Parallel Lisp". In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, Portland, OR, June 21–23, 1989.

[61] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Conference on Supercomputing*, pages 195–206, July 1993.

[62] V. Lanin and D. Shasha. "A Symmetric Concurrent B-Tree Algorithm". In *Proceedings of the 1986 Fall Joint Computer Conference*, pages 380–386, Dallas, TX, November 2–6, 1986.

[63] P.L. Lehman and S.B. Yao. "Efficient Locking for Concurrent Operations on B-Trees". *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[64] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S. Yang, and R. Zak. "The Network Architecture of the Connection Machine CM-5". *Journal of Parallel and Distributed Computing*, to appear. An early version appeared in the *Proceedings of the 1992 Symposium on Parallel Architectures and Algorithms*.

[65] 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 International Symposium on Computer Architecture*, pages 148–158, Seattle, WA, May 1990.

[66] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[67] K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems". *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[68] B. Liskov, M. Day, and L. Shrira. "Distributed Object Management in Thor". In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, 1993.

[69] C.R. Manning. "ACORE: The Design of a Core Actor Language and its Compiler". Master's thesis, Massachusetts Institute of Technology, August 1987.

[70] E. Markatos. *Scheduling for Locality in Shared-Memory Multiprocessors*. PhD thesis, University of Rochester, 1993.

[71] E.P. Markatos and T.J. LeBlanc. "Load Balancing vs. Locality Management in Shared-Memory Multiprocessors". Technical Report 399, University of Rochester Computer Science Department, October 1991.

[72] P.R. McJones and G.F. Swart. "Evolving the UNIX System Interface to Support Multi-threaded Programs". Technical Report 21, Digital Systems Research Center, Paul Alto, CA, September 28, 1987.

[73] W. Mendenhall. *Introduction to Probability and Statistics*. Duxbury Press, North Scituate, MA, fifth edition, 1979.

[74] Y. Mond and Y. Raz. "Concurrency Control in B+ Trees Using Preparatory Operations". In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 331–334, Stockholm, Sweden, August 21–23, 1985.

[75] F. Mueller. "A Library Implementation of POSIX Threads under UNIX". In *Proceedings of 1993 Winter USENIX*, pages 29–41, San Diego, CA, January 1993. Available at ftp.cs.fsu.edu:/pub/PART/pthreads.tar.Z.

[76] R.S. Nikhil. "*Cid*: A Parallel, "Shared-memory" C for Distributed-memory Machines". In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 376–390, Ithaca, NY, August 8–10, 1994. Springer-Verlag.

[77] P. Pegnato. Personal communication, June 1995. Authorization to use Monotype Typograph's description of Monotype Garamond.

[78] 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, August 20–23, 1985.

[79] M.L. Powell and B.P. Miller. "Process Migration in DEMOS/MP". In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 110–119, Bretton Woods, NH, October 10–13, 1983.

[80] M.C. Rinard, D.J. Scales, and M.S. Lam. "Jade: A High-Level, Machine-Independent Language for Parallel Programming". *IEEE Computer*, 26(6):28–38, June 1993.

[81] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. "Supporting Dynamic Data Structures on Distributed-Memory Machines". *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[82] Y. Sagiv. "Concurrent Operations on B-Trees with Overtaking". *Journal of Computer and System Sciences*, 33(2):275–296, October 1986.

[83] H.S. Sandhu, B. Gamsa, and S. Zhou. "The Shared Regions Approach to Software Cache Coherence on Multiprocessors". In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, pages 229–238, San Diego, CA, May 19–22, 1993.

[84] D.J. Scales and M.S. Lam. "The Design and Evaluation of a Shared Object System for Distributed Shared Memory Machines". In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 101–114, Monterey, CA, November 14–17, 1994.

[85] K.E. Schauser, D.E. Culler, and T. von Eicken. “Compiled-Controlled Multithreading for Lenient Parallel Languages”. In *Proceedings of the 5th ACM Conference on Functional Languages and Computer Architecture*, Cambridge, MA, August 26–30, 1991.

[86] SPARC International, Inc., Menlo Park, CA. *The SPARC Architecture Manual*, version 8 edition, 1992.

[87] M.S. Squillante and R.D. Nelson. “Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling”. Technical Report 90-07-05, University of Washington Department of Computer Science, September 1990.

[88] J.W. Stamos and D.K. Gifford. “Remote Evaluation”. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[89] P. Stenström, M. Brorsson, and L. Sandberg. “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing”. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 109–118, San Diego, CA, May 16–23, 1993. Cox and Fowler [30] have a similar algorithm.

[90] D. Stodolsky, J.B. Chen, and B.N. Bershad. “Fast Interrupt Priority Management in Operating System Kernels”. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 105–110, San Diego, CA, September 20–21, 1993.

[91] K. Taura. “Design and Implementation of Concurrent Object-Oriented Programming Languages on Stock Multicomputers”. Master’s thesis, University of Tokyo, February 1994.

[92] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite Jr. “Firefly: A Multiprocessor Workstation”. In *Proceedings of the 2nd Conference on Architectural Support for Programming Languages and Systems*, Palo Alto, CA, October 5–8, 1987.

[93] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, version 3.0 edition, May 1993.

[94] T. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors.* PhD thesis, University of California at Berkeley, 1993.

[95] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. “Active Messages: a Mechanism for Integrated Communication and Computation”. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 19–21, 1992.

[96] D.W. Wall. “Register Windows vs. Register Allocation”. In *Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation*, pages 67–78, Atlanta, GA, June 22–24, 1988.

[97] D.A. Wallach, W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, and W.E. Weihl. “Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation”. In *Proceedings of*

the 5th Symposium on Principles and Practice of Parallel Programming, pages 217–226, Santa Barbara, CA, July 1995.

[98] N. Walsh. “Internet Font Archive”. http://jasper.ora.com/comp.fonts/ifa/ifa.cgi.

[99] P. Wang. “An In-Depth Analysis of Concurrent B-Tree Algorithms”. Master’s thesis, Massachusetts Institute of Technology, January 1991. Available as MIT/LCS/TR-496.

[100] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. “PRELUDE: A System for Portable Parallel Software”. Technical Report MIT/LCS/TR-519, MIT Laboratory for Computer Science, October 1991. A shorter version appears in *Proceedings of 4th International Conference on Parallel Architectures and Languages*.

[101] A.C. Yao. “On Random 2–3 Trees”. *Acta Informatica*, 9:159–170, 1978.

[102] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. “Software Write Detection for a Distributed Shared Memory”. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, Monterey, CA, November 14–17, 1994.

[103] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. “Vienna Fortran — a Language Specification”. Technical Report ICASE Interim Report 21, ICASE NASA Langley Research Center, March 1992.

[104] Q.Y. Zondervan. “Increasing Cross-Domain Call Batching using Promises and Batched Control Structures”. Master’s thesis, Massachusetts Institute of Technology, June 1995.

About the Author

The author matriculated at the Massachusetts Institute of Technology in the fall of 1984. He participated in the VI-A internship program, through which he worked at IBM T.J. Watson Research Center on his master's degree; he received his bachelor's and master's degrees in May 1988. He then decided that he had not had enough of the 'Tute, and that he just had to acquire one more degree there. As a result, he stayed in Cambridge to round out over a decade of schooling at MIT. He was generously supported by a National Science Foundation Graduate Fellowship for three years, during which time he managed to avoid graduating. He is a co-author of two journal papers, eight conference papers, several workshop papers, two turtledoves, and, most importantly, one dissertation. After completing this dissertation (and realizing that there are other places than MIT), he will begin work as a postdoctoral researcher at the University of Washington.

The author is a member of Tau Beta Pi, Phi Beta Kappa, Sigma Xi, the IEEE Computer Society, and ACM. In his spare time, the author enjoys playing his viola (particularly chamber music), as well as playing ultimate and squash.

Colophon

This text of this document was prepared using `emacs` version 18.58.5, which was written by Richard Stallman and the Free Software Foundation. Figures were drawn using `idraw`, which was written by the Interviews group at Stanford University. Graphs were produced using `jgraph`, which was written by Jim Plank. Most of the data for the graphs was processed using `perl` version 4.0, which was written by Larry Wall.

This document was typeset using LaTeX version 2.09, a document preparation system written by Leslie Lamport; LaTeX is based on TeX, a typesetting system written by Donald Knuth. The fonts used are Monotype Garamond for the main text, Courier Narrow (a version of Courier that has been artificially condensed by 80%) for fixed-width text, and Times Italic and Symbol for mathematics. Many thanks are due to Ed Kohler for his invaluable help in choosing and setting up these fonts.

Monotype Garamond was designed by Monotype Type Drawing Office in 1922. This typeface is based on roman types cut by Jean Jannon in 1615. Jannon followed the designs of Claude Garamond which had been cut in the previous century. Garamond's types were, in turn, based on those used by Aldus Manutius in 1495 and cut by Francesco Griffo. The italic is based on types cut in France circa 1557 by Robert Granjon. Garamond is a beautiful typeface with an air of informality which looks good in a wide range of applications. It works particularly well in books and lengthy text settings.

The description of Monotype Garamond is Copyright (C) 1994, 95 by Monotype Typography, Inc. [98]. All Rights Reserved. Used here with permission of Monotype Typograph, Inc. [77].