Multigrain Shared Memory

by

Donald Yeung

B.S., Computer Systems Engineering
Stanford University, 1990

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1993

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

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1998

Signature of Author: ______________________________

Department of Electrical Engineering and Computer Science
December 17, 1997

Certified by: ______________________________

Anant Agarwal
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: ______________________________

Arthur C. Smith
Chairman, Departmental Graduate Committee

Multigrain Shared Memory

by

Donald Yeung

Submitted to the Department of Electrical Engineering and Computer Science
on December 17, 1997 in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy
in Electrical Engineering and Computer Science

ABSTRACT

Parallel workstations, each comprising a 10-100 processor shared memory machine, promise cost-effective general-purpose multiprocessing. This thesis explores the coupling of such small- to medium-scale shared memory multiprocessors through software over a local area network to synthesize larger shared memory systems. Multiprocessors built in this fashion are called Distributed Scalable Shared memory Multiprocessors (DSSMPs).

The challenge of building DSSMPs lies in seamlessly extending hardware-supported shared memory of each parallel workstation to span a cluster of parallel workstations using software only. Such a shared memory system is called *Multigrain Shared Memory* because it naturally supports two grains of sharing: fine-grain cache-line sharing within each parallel workstation, and coarse-grain page sharing across parallel workstations. Applications that can leverage the efficient fine-grain support for shared memory provided by each parallel workstation have the potential for high performance.

This thesis makes three contributions in the context of Multigrain Shared Memory. First, it provides the design of a multigrain shared memory system, called MGS, and demonstrates its feasibility and correctness via an implementation on a 32-processor Alewife machine. Second, this thesis undertakes an in-depth application study that quantifies the extent to which shared memory applications can leverage efficient shared memory mechanisms provided by DSSMPs. The thesis begins by looking at the performance of unmodified shared memory programs, and then investigates application transformations that improve performance. Finally, this thesis presents an approach called *Synchronization Analysis* for analyzing the performance of multigrain shared memory systems. The thesis develops a performance model based on Synchronization Analysis, and uses the model to study DSSMPs with up to 512 processors. The experiments and analysis demonstrate that scalable DSSMPs can be constructed from small-scale workstation nodes to achieve competitive performance with large-scale all-hardware shared memory systems. For instance, the model predicts that a 256-processor DSSMP built from 16-processor parallel workstation nodes achieves equivalent performance to a 128-processor all-hardware multiprocessor on a communication-intensive workload.

Thesis Advisor: A. Agarwal
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

Pursuing my doctorate at MIT has been the most challenging endeavor I have ever undertaken. In retrospect, I definitely appreciate the opportunity I've been given to face such a challenge because of the enrichment I have received both in my professional and personal life in ways I could never have imagined prior to arriving in Cambridge seven and a half years ago. For these life-changing experiences, I owe many thanks to a number of people.

I would like to begin by acknowledging my advisor, Anant, for his invaluable guidance which has kept me going all these years, and for his never-ending moral support, without which I would never have had the confidence to overcome the hurdles of my degree. I will forever be in awe of his ability to think on his feet (a.k.a. "shoot from the hip"), his infinite energy, and his ability to get others excited about almost anything. Most importantly, I will always hold the greatest respect for Anant because of his commitment to promoting his students before himself so that they may succeed.

I would also like to acknowledge the members of the Alewife team, without whom none of my thesis would have been possible. Thanks go to Beng, Dan, David (Chaiken), David (Kranz), and Rajeev, to whom I owe all the simulation, compilation, debugging, and runtime software infrastructure in the Alewife machine. To Kirk, I owe many inspiring technical conversations, particularly early on in my career at MIT. Thanks go to Fred for all his advice on interviews that allowed me to survive my job search. To Anne, I (as well as everyone else in the Alewife group) owe all the support that kept the group from an administrative train wreck. To Ken, perhaps the most compatible office mate I will ever have, I owe many lessons on hardware hacking, as well as many stimulating research discussions, some that lead to important ideas for my thesis. Of course, I will be eternally grateful to Kubi, who managed to be a colleague, mentor, and friend all at the same time, and whose tremendous knowledge and wisdom make him one of the greatest teachers I will ever have. And together, to Kubi and Ken, I owe the endless hours of comradary that have defined the "Alewife experience" for me, and that I will remember for the rest of my life.

Finally, my greatest thanks go to Mom, Dad, Marina, and Melany, whose love and support has allowed me to leave MIT with my sanity intact.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large-scale shared memory multiprocessors have received significant attention within the computer architecture community over the past decade. The high interest that these architectures have generated is due in large part to their cost-performance characteristics: large-scale shared memory multiprocessors have the potential to deliver supercomputer performance at commodity server costs.

Large-scale shared memory machines offer the promise of remarkable levels of cost-performance because they are constructed from computational building blocks, or compute nodes, that require only modest technology. Fueled by the microprocessor, such compute nodes individually deliver reasonable performance while using commodity computational technology. Supercomputer performance is achieved on these systems by coupling multiple compute nodes together to take advantage of medium- to coarse-grain parallelism. While the architecture does not assist in the discovery of parallelism, provided enough parallelism can be identified by a compiler or a programmer to keep all compute nodes busy, high performance can be sustained.

While the promise of remarkable cost-performance has made large-scale shared memory architectures attractive, thus far, this promise has gone unfulfilled. In practice, the potential cost-performance benefits promised by large-scale shared memory architectures are difficult to realize because of the tension between providing efficient communication mechanisms and maintaining cost-efficiency at large scales. Because large-scale shared memory architectures rely on parallelism to achieve performance, they must be equipped to support the communication that arises when the computational load represented by a single application is distributed across multiple computational elements. Therefore, in addition to providing per-node computational throughput, an equally (if not more) important architectural requirement is to provide efficient communication mechanisms. Without efficient communication, parallel applications with demanding communications requirements cannot be supported on these architectures, thus limiting the scope of problems for which these architectures can be effectively applied.

Traditionally, large-scale shared memory multiprocessors provide efficient communication mechanisms through aggressive architectural support. An example is the hardware cache-coherent distributed shared memory (DSM) architecture. Hardware DSMs are built using custom communication interfaces, high performance VLSI interconnect, and special-purpose hardware support for shared memory. These aggressive architectural features provide extremely efficient communication support between nodes through tightly coupled hardware interfaces. The architectural support for efficient shared memory communication allows hardware DSMs to provide scalable performance even on communication-intensive applications.

While aggressive architectural support leads to high performance, the investment in hardware mechanisms comes at a cost. In particular, tight coupling between nodes is difficult to maintain in a cost-effective manner as the number of nodes becomes large. Fundamental obstacles prevent large tightly-coupled systems from being cost effective. The cost of power distribution, clock distribution, cooling, and special packaging considerations in tightly coupled systems do not scale linearly with size. As system size is scaled, cost grows disproportionately due to the physical limitations imposed by the necessity to maintain tight coupling across all the nodes in the system. Perhaps most important, the large-scale nature of these machines prevents them from capitalizing on the economy of cost that high volume smaller-scale machines enjoy.

In response to the high design cost of large-scale hardware DSMs, many researchers have proposed building large-scale shared memory systems using commodity uniprocessor workstations as the compute node building block. In these lower cost systems, the tightly coupled communications interfaces found in hardware DSMs are replaced by commodity interfaces that do not require any special-purpose hardware. Furthermore, commodity networks such as those found in the local area environment are used to connect the workstation nodes, and the shared memory communication abstraction is supported purely in software. Such software DSM architectures are cost effective because all the components are high volume commodity items and because specialized tightly-coupled packaging is not required.

Unfortunately, software DSMs are unable to provide high performance across a wide range of applications. While communication interfaces for commodity workstations have made impressive improvements, the best reported inter-workstation latency numbers are still an order of magnitude higher than for machines that have tightly-coupled special-purpose interfaces [66]. Furthermore, the best latencies come from special networks that do not have the large volume required for low cost commoditization. The higher cost of communication on commodity systems prevents them from supporting applications with intensive communication requirements.

Existing architectures for large-scale shared memory machines have not satisfactorily addressed the tension between providing efficient communication mechanisms for high performance and designing for cost effectiveness. In this thesis, we propose a novel approach to building large-scale shared memory machines that offers higher cost-performance properties than existing architectures. Our approach leverages the scalable shared memory multiprocessor (SSMP) as the building block for larger systems.

SSMP is a general name for any small- (2–16 processors) to medium-scale (17–128 processors) shared memory machine. A familiar example is the bus-based Symmetric Multiprocessor (SMP). Another example is the small- to medium-scale distributed-memory multiprocessor. The latter architecturally resembles large-scale (greater than 128 processors) tightly-coupled machines, but is targeted for smaller systems. The general nature of the SSMP terminology suggests that both SMPs and distributed-memory architectures are suitable building blocks for larger shared memory systems.

The SSMP is an attractive building block for large-scale multiprocessors for two reasons. First, SSMPs provide efficient hardware support for shared memory. A larger system that can leverage this efficient hardware support has the potential for higher performance than a network of conventional uniprocessor workstations in which shared memory is implemented purely in software. And second, the efficient shared memory mechanisms provided by SSMPs do not incur exorbitant costs because the tight coupling required is only provided across a small number of processors. Unlike large-scale hardware DSMs, small-scale tightly-coupled systems can be cost-effective, as evidenced by the commodity nature of the SMP architecture.

We call a large-scale system built from a collection of SSMPs a *Distributed Scalable Shared memory Multiprocessor* (DSSMP). DSSMPs are constructed by extending the hardware-supported shared memory in each SSMP using software distributed shared memory (DSM) techniques to form a single shared memory layer across multiple SSMP nodes. Such hybrid hardware-software systems support shared memory using two granularities, hence the name *Multigrain Shared Memory*. Cache-coherent shared memory hardware provides a small cache-line sharing grain between processors colocated on the same SSMP. Page-based software DSM provides a larger page sharing grain between processors in separate SSMPs.

1.1 Contributions

This thesis presents a thorough investigation of multigrain shared memory architectures. The fundamental idea researched by this thesis is the coupling of small- to medium-scale shared memory multiprocessors using software shared memory techniques to build a large-scale system. The challenge lies in synthesizing a single transparent and seamless shared memory layer through the cooperation of both fine-grain hardware cache-coherent and coarse-grain page-based shared memory mechanisms. The thesis reports on an extensive systems building experience that has taken the basic notion of multigrain shared memory and carried it through an exhaustive systems investigation project.

The aggregate contributions from our investigation of multigrain shared memory cover every phase of a complete systems building and evaluation process. Contributions are made in system design, which include a complete set of mechanisms that enable the construction of multigrain shared memory. From the design, our investigation has built a prototype implementation that demonstrates the feasibility and correctness of our design. Using the prototype implementation, we conduct an in-depth evaluation that experimentally characterizes the behavior of the system. Finally, our investigation also includes an analysis phase that tries to provide insight into why the system behaves the way it does.

The specific contributions of this thesis are summarized below, organized into three categories: design and implementation, evaluation, and analysis.

Design and Implementation

- This thesis investigates the design of multigrain shared memory systems. Fundamental to the design are a set of mechanisms that enable the cooperation of hardware cache-coherent and software page-based shared memory layers. Three key mechanisms are identified that enable such cooperation: multiprocessor VM faults, TLB coherence, and page cleaning.

- In addition to fundamental design issues, this thesis also investigates how to build fast multigrain shared memory systems. For performance, multigrain architectures must export the efficiency of the hardware cache-coherent mechanisms as often as possible. When application sharing patterns permit, the system should remove any intervention from the software layers until software services are absolutely necessary. The thesis proposes the Single-Writer mechanism that achieves this design requirement.

- A complete multigrain shared memory system design is proposed, called MGS. MGS integrates the multigrain-specific mechanisms described above along with conventional software shared memory mechanisms to construct a fully functional multigrain architecture. To our knowledge, MGS is the first system to facilitate a comprehensive study of multigrain shared memory.

- A prototype of the MGS design is implemented on a 32-processor Alewife machine. The prototype demonstrates the correctness of the design, and provides a platform for experimentation. A key feature of the prototype is *virtual clustering*, which allows the clustering configuration of the DSSMP to be changed at runtime. This flexibility enables the evaluation of different DSSMP configurations, a crucial capability leveraged by the experimental methodology of this thesis.

Evaluation

- The thesis presents a performance framework that characterizes application behavior on multigrain systems. The framework measures the sensitivity of application performance to varying cluster configurations using two performance metrics: *Multigrain Potential*, and *Breakup Penalty*. Together, these metrics report how an application responds to different mixes of fine-grain and coarse-grain shared memory support, and provides calibration of an application's performance on multigrain systems against its performance on all-software and all-hardware systems.

- An in-depth experimental evaluation is conducted on the MGS prototype. Several micro-benchmarks along with 9 shared memory applications representing a wide range of scientific workloads are studied.

- An extension to the application study performed above is undertaken to understand the performance bottlenecks encountered by the applications. Transformations to relieve these bottlenecks are applied manually, and the improvements in performance are measured. The extended application study shows the potential performance that multigrain systems can deliver when additional compiler and/or programmer effort is applied to off-the-shelf applications. The study includes a qualitative evaluation of each transformation's sophistication. In the process, those transformations that can be performed automatically by existing compilers are identified.

- The sensitivity of application performance on MGS to inter-SSMP communication latency and page size (the granularity of coherence between SSMPs) is studied.

Analysis

- A novel approach to analyzing performance on software shared memory systems is proposed, called *Synchronization Analysis*. Synchronization analysis enables the prediction of shared memory communication volume through analysis of a program's synchronization dependence graph. This synchronization-centric technique relies on the insight that communication patterns are highly correlated with synchronization patterns in software shared memory systems. Synchronization analysis represents a shift from traditional program analysis techniques used by parallel optimizing compilers that are data-centric in nature.

- The thesis presents a performance model based on synchronization analysis that predicts execution time on multigrain shared memory systems. Model parameters for the MGS prototype are determined to enable prediction of application performance on MGS. The accuracy of the model is validated by comparing model predictions using the MGS model parameters against experimental measurements taken on the MGS prototype.

- Using the performance model developed for MGS, an analytic study is conducted to evaluate the scalability of the MGS system. Both problem size and machine size are scaled beyond what can be studied experimentally. Machines of up to 512 processors are evaluated.

1.2 Outline

This section briefly outlines the contents of the thesis. Chapters 2 and 3 provide introductory discussion for the rest of the thesis. Chapter 2 provides background material that forms the foundation for the work reported in this thesis. Concepts from conventional hardware and software distributed shared memory systems are reviewed. Chapter 3 discusses the impact of sharing granularity supported by a shared memory architecture on both application performance and on system cost. The chapter argues that there exists a tension between supporting fine-grain applications efficiently on large-scale machines, and designing large-scale machines in a cost-effective manner. This tension is the motivation for what we call the *multigrain approach* to designing shared memory systems, which is the foundation for all the ideas presented in this thesis.

Following the introductory chapters, Chapters 4 and 5 present the meat of the system proposed in this thesis, called MGS. Chapter 4 presents the architecture of the MGS system. It first describes the mechanisms necessary to build multigrain shared memory and to make it efficient. The chapter includes fairly detailed discussion on the exact mechanics that support the architecture. While Chapter 4 describes the MGS design, Chapter 5 describes an actual prototype we have built on the Alewife multiprocessor platform. Chapter 5 begins by explaining the *virtual clustering* approach that is central to our implementation (and to the evaluation later on). The chapter covers many implementation issues that arise when implementing MGS on Alewife.

The next two Chapters, 6 and 7, evaluate the performance of the MGS system. Chapter 6 presents the experimental portion of our evaluation that uses both microbenchmarks and a full range of shared memory applications. The behavior of applications is studied both using the original applications in their "off-the-shelf" form, and when transformations are applied to improve their locality properties. Chapter 7 presents the analytic portion of our evaluation. Much of the chapter is devoted to the presentation of a novel approach for performance analysis called *Synchronization Analysis*. Then, a performance model based on synchronization analysis is presented and validated against our MGS prototype. Using the performance model, we study the scalability of the MGS system on one of the applications from the experimental study, scaling machine size up to 512 processors.

Finally, Chapter 8 discusses related work, and Chapter 9 closes the thesis with conclusions.

Chapter 2

Background

This chapter describes the concept of distributed shared memory, along with various issues concerning the implementation of distributed shared memory systems. Section 2.1 introduces the notion of distributed shared memory, including a discussion on the cache coherence problem and how it is solved on distributed shared memory machines. And Section 2.2 presents several issues that concern hardware and software implementations of distributed shared memory. The information provided in this chapter forms the basis for the work presented in the rest of this thesis. Those readers familiar with distributed shared memory are encouraged to continue reading in Chapter 3.

2.1 Distributed Shared Memory

Shared memory is a programming model in which multiple threads of control communicate with one another through a single transparent layer of logically shared memory, as illustrated by Figure 2.1. It has been argued that shared memory is a desirable programming model since communication happens implicitly each time two threads access the same memory location. This is in contrast to a message passing programming model where the responsibility of managing communication is placed explicitly on the programmer [41].

Shared memory multiprocessors implement the shared memory programming model by supporting the shared memory abstraction directly in the system architecture. An example shared memory architecture for which the design faithfully resembles the programming model is the Symmetric Multiprocessor (SMP). SMPs implement the shared memory programming model by connecting multiple processors directly to physical memory through a single memory controller. The connection fabric that allows processors to communicate with the memory controller is a shared bus. Shared memory is supported by the fact that processors share the same image of physical memory.

While the architecture of an SMP directly implements the shared memory programming model, it is not suitable for large-scale shared memory systems because it is not *scalable*. The bus interconnect and single physical memory image become performance bottlenecks as the number of processors inside the SMP is increased. At some scaling

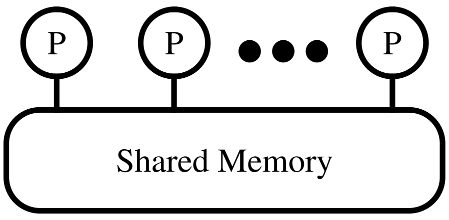


Figure 2.1: Shared memory multiprocessors provide a single transparent view of memory across all processors.

point, the communication bandwidth between processors and memory saturates resulting in the serialization of concurrently issued shared memory transactions, and performance degradation. To address the resource contention problems that limit scalability in the SMP architecture, it is necessary to replace the serial communication and memory interfaces with parallel interfaces. This is the goal of the distributed shared memory multiprocessor.

2.1.1 Addressing Resource Contention

Distributed shared memory (DSM) architectures solve the resource contention problem by distributing physical memory. In a DSM, the single logical shared memory address space is partitioned across multiple physical memory modules, and each memory module is given its own dedicated memory controller that services shared memory requests destined to that module. A shared memory *abstraction* is synthesized across the physically distributed memories via shared memory modules, one per physical memory and controller, that communicate using point-to-point messages across a switched interconnection network, such as those discussed in [18]. Figure 2.2 illustrates these components that make up the DSM architecture. In the figure, each processor, its local memory, and its local shared memory module together form a *DSM node*.

Synthesis of a shared memory abstraction in a DSM occurs in the following manner. When a shared memory module receives a shared memory request from a local processor, it determines the physical memory module for which the request is destined using a mapping that reflects the partitioning of logical shared memory across the physical memory modules. If the request maps to the local physical memory module, the request is satisfied immediately through local memory. If the request maps to a remote physical memory module, the shared memory module initiates a remote transaction by sending a message that contains the desired shared memory address across the interconnection network to the appropriate remote shared memory module. The remote shared memory module responds to the transaction by accessing the shared memory location in its local physical memory module, and then sending the data back to the requesting shared memory module in another message. The transaction completes when this data is supplied to the requesting processor.

The distribution of physical memory and memory interfaces in a DSM allow the aggre-

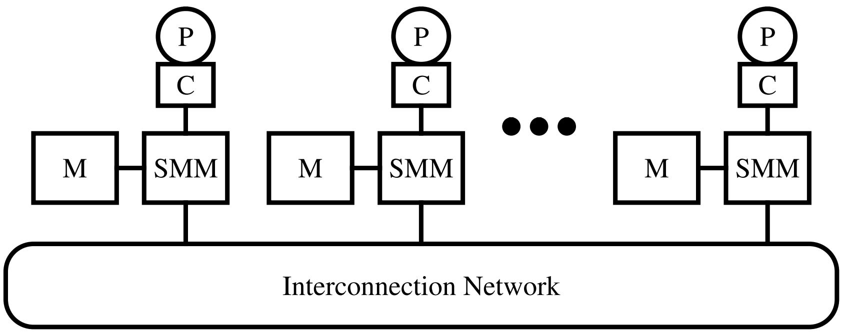


Figure 2.2: DSMs partition the logical shared address space across multiple physical memory modules (M), and synthesize a shared memory abstraction via shared memory modules (SMM) that communicate across an interconnection network. Each shared memory module services memory requests from a processor (P); shown with each processor is a hardware cache (C). Together, a processor (and its hardware cache), its local memory, and its local shared memory module form a DSM node.

gate memory bandwidth to scale along with the number of processing elements. When the size of a DSM is increased, not only are processing elements added, but with the additional processors, physical memory modules and memory controllers are added as well. Furthermore, the amount of communication bandwidth supplied between memory modules can be increased by adding switches to grow the size of the interconnection network. Because of the ability to scale processing, memory, and communication resources together (as opposed to scaling processing resources alone as was the case in the SMP architecture), the DSM provides scalability.

2.1.2 Cache Coherence

Caching replicates data across a memory hierarchy from slower storage into faster storage. The goal is to keep the most frequently accessed data in the highest level of the memory hierarchy (fastest storage) so that it can be accessed efficiently by the processor. For those applications that demonstrate memory access locality, caching can very effectively reduce the overheads associated with memory operations performed by the processor.

The implementation of caching in shared memory multiprocessors leads to the well-known *cache-coherence problem*. In a multiprocessor that permits the caching of shared data, it is possible for the data associated with a single shared memory address to become replicated in multiple processor caches. If a processor tries to write a new value into one cached copy, the other cached copies will become incoherent or stale with respect to the written copy.

The cache coherence problem in shared memory multiprocessors is addressed by maintaining coherence on cached data using a *cache-coherence protocol*. DSMs typically employ cache-coherence protocols that are *directory based* [15, 32, 3, 14, 63]. Directory-based cache-coherence protocols maintain a directory entry for each cache block of data in shared memory. Each time a request for a cache block is fulfilled by the shared memory module, the ID of the DSM node from which the request originated is recorded in the associated directory entry. Therefore, the directory entry for a cache block records the set of nodes which have a copy of the cache block at any given moment in time.

Coherence can be maintained on a particular cache block by using the directories to perform *invalidation*. The process of invalidation is initiated by the shared memory module where the directory entry resides, typically called the cache block's *home node*. At the home node, the directory entry for the cache block is consulted by the shared memory module, and an invalidation message is sent to every DSM node specified in the directory entry. At each node, the invalidation message causes the copy of the cache block to be purged from the cache. The cache then sends an acknowledgment message back to the home node. If the purged cache block is dirty, the acknowledgment message also includes data that reflects the updates performed on the cache block. At the home, the shared memory module processes the acknowledgments, and any data in acknowledgments with updates are merged into the location in the memory module that provides the backing for the cache block (see Sections 2.2.1 and 2.2.2 for more details on updates and how they are merged). After merging the update(s), the data in the memory module reflects the most recent version of the data as a result of the writes performed on the cache block up to the point when invalidation was initiated.

Invalidation maintains coherence for two reasons. First, it provides a means for updates performed at the caches to propagate back to the home. This allows subsequent requests after an invalidation to receive data that reflects the updates. Second, it is the mechanism by which stale data is reclaimed. Each invalidation removes data that has become stale in a processor's cache. Subsequent accesses performed to an invalidated cache block will therefore not access the stale value, but instead will re-request the cache block from the home and receive an updated value.

2.2 DSM Implementation

Section 2.1 above discussed distributed shared memory and caching in DSMs as general concepts. In this section, we take a closer look at two specific DSM implementations, the hardware cache-coherent DSM (examples include [46, 44, 23, 38, 47]), and the software page-based DSM (examples include [48, 6, 13, 36, 37]). We will focus on how the implementation of distributed shared memory and cache coherence differ on these two architectures.

The primary difference between hardware and software DSMs is the level in the memory hierarchy on each DSM node where caching is performed. In the hardware DSM, caching is performed in the hardware processor caches, and in the software DSM, caching is performed in main memory (the "C" and "M" modules, respectively, in Figure 2.2). Where caching occurs determines the *coherence unit*, a block of memory which the DSM treats as an indivisible unit of data during replication and invalidation. Because hardware DSMs perform caching in processor caches, its coherence unit is the processor cache line.

Similarly, because the software DSM performs caching in main memory, its coherence unit is the page. The size or *granularity* of the coherence unit is a crucial system parameter, and is the topic of Sections 3.2 and 3.3 in Chapter 3.

In the next two sections, we discuss the implications for performing caching in either hardware caches or main memory.

2.2.1 Hardware Cache Coherence

Hardware DSMs provide special-purpose hardware support for shared memory (*i.e.* the shared memory modules in Figure 2.2 are implemented in hardware). This shared memory hardware synthesizes a physical shared memory address space, and maintains coherence on the shared data cached in the processor caches, as described in Section 2.1 above.

The shared memory hardware is integrated with the mechanisms that manage the processor caches through cache miss events. When the processor on a DSM node suffers a cache miss, the cache miss is intercepted by the shared memory hardware on the processor's local node. Based on the address of the cache miss, the shared memory hardware either fulfills the cache miss directly from local memory, or from remote memory via communication with a remote shared memory module. In the latter case, the creation and transmission of messages across the interconnection network, as described in Section 2.1.1, is performed purely in hardware.

The shared memory hardware not only supports the shared memory abstraction, but it also supports cache coherence. For simplicity, hardware DSMs typically implement a *single-writer protocol* using an *invalidate-on-write* policy (simplicity of the cache-coherence protocol is important due to the complexities of hardware implementation). In a single-writer protocol, multiple outstanding cache copies are allowed so long as accesses to the copies are confined to reads. Once a processor tries to perform a write, the protocol invalidates all outstanding copies. Single-writer protocols are simple because there can be at most one outstanding cache block that is dirty at any given time. Therefore, when a dirty cache block is invalidated, the entire cache block can be written into the memory module at the home node thus propagating any updates performed on the cache block. Supporting multiple writers is more complex because it requires multiple dirty cache blocks to be merged during invalidation. In Section 2.2.2, we discuss the protocol additions necessary to support multiple writers.

While single-writer protocols are simple and therefore desirable from a hardware implementation standpoint, they can introduce performance penalties. In particular, each shared memory write could potentially cause the invalidation of one or more cache copies[1]. If the processor performing such a write is stalled for the duration of the invalidation(s), significant cache miss stall can be introduced, thus degrading performance. Many techniques have been proposed to address such cache miss stall overhead, such as software-controlled prefetching [51], block multithreading [43, 68], and relaxed memory

[1] In the worst case, there could be $P-1$ outstanding copies, where P is the number of DSM nodes.

consistency models [1, 25, 19]. We discuss relaxed memory consistency models below; a discussion and evaluation of all three techniques appears in [27].

Relaxed Memory Consistency

The *memory consistency model* supported by a shared memory multiprocessor determines the ordering in which individual processors view the memory operations performed by other processors in the system. The strongest form of memory consistency is *sequential consistency* [45]. Sequential consistency states that a total order can be defined across all shared memory operations, and furthermore, all processors see the same total order. Multiprocessors that stall a processor on each shared memory write until all outstanding cache copies are invalidated support sequential consistency. Stalling writes during invalidation ensures that when a write is performed, no other copies of the cache line exist in the system. Therefore, there is a well-defined moment in time relative to all processors that the write commits, a condition necessary for sequential consistency.

Higher performance can be attained if the system is not required to support such a singular commit point for writes. Memory consistency models that are less strict than sequential consistency in their guarantees on event ordering, known as relaxed memory consistency models, allow systems to overlap the overhead of maintaining consistency on shared data with useful computation. Instead of guaranteeing that the updates performed by a processor are visible to all other processors after every shared memory write (as is provided by sequential consistency), relaxed memory consistency only guarantees that updates are visible at special points in a program. An example of such a relaxed memory model is *release consistency* (RC) [25]. Programs written assuming an RC memory consistency model must include special annotations, known as *releases* and *acquires*, that specify to the memory system when coherence is necessary. The memory system guarantees strict ordering between release and acquire operations (*i.e.* releases and acquires are sequentially consistent), but individual shared memory operations between releases and acquires can reorder.

Relaxing the order in which individual processors view the updates performed across all processors enables the complete elimination of memory stall due to cache misses on writes. Because a singular commit point is not needed under RC, a processor performing a write can issue the write immediately without being stalled even if the location being written has been cached by other processors. Each write performed by a processor is buffered, usually in a hardware buffer provided by the processor[2]. The shared memory hardware propagates the updates in the write buffer at the pace of the memory system which can be slow if invalidations are required. Meanwhile, the overhead of maintaining coherence on the buffered updates is hidden from the processor because the processor is allowed to continue performing useful computation. Notice this breaks sequential consistency. For example, if two writes are performed simultaneously to the same location in shared memory, each processor will think its write happened first.

[2]Most modern processors provide on-chip write buffers.

2.2.2 Page-Based DSMs

Software DSMs rely purely on software to support shared memory (*i.e.* the shared memory modules in Figure 2.2 are implemented in software). Therefore, there is no sharing across physical addresses as there is in hardware DSMs. The memory modules on different DSM nodes have separate physical address spaces. Instead, the shared memory abstraction is constructed by enforcing a single shared virtual address space across DSM nodes.

A shared virtual address space is constructed by leveraging the address translation and protection checking mechanisms as is commonly provided by hardware TLBs (Translation Lookaside Buffers). The address translation mechanism provides the level of indirection needed between shared virtual memory and physical memory so that each DSM node's physical memory can be used as a cache for replicated pages. The protection checking mechanism provides access control on processor accesses so that software shared memory can be invoked for those processor accesses which require distributed shared memory service. The software shared memory module is integrated with the TLB fault handler. TLB faults to pages under software distributed shared memory management are passed to the shared memory module for processing. Therefore, protection checking as provided by TLBs serves the same purpose that cache misses do for hardware DSMs–it is the conduit through which shared memory requests get forwarded from the processor to the shared memory module.

Compared to hardware DSMs, software DSMs are more sensitive to frequent inter-node communication for two reasons. First, anytime communication occurs, the shared memory module must intervene, and software implementation of the shared memory module is less efficient than hardware implementation. Second, messaging across the inter-node network in Figure 2.2 is expensive for page-based systems because software DSMs usually employ commodity local area networks (LANs) and commodity network interfaces as compared to the VLSI networks and specialized interfaces used in hardware DSMs. Consequently, it is especially important to minimize the amount of inter-node communication for software DSMs. Below, we discuss two techniques that together reduce communication in software DSMs: delayed updates and multiple-writer protocols.

Delayed Updates

As described in Section 2.2.1 above, the RC memory consistency model allows coherence overhead to be overlapped with useful computation by relaxing the guarantees on event ordering. This permits the updates performed by a processor to be decoupled from the overhead of maintaining coherence on those updates, thereby eliminating stalls due to writes. While this relieves the processor from coherence overheads, it does not relieve the shared memory system and the interconnection network between DSM nodes from such overheads since the coherence of the updates must be enforced eventually.

The relaxation of event ordering guarantees permitted by RC can be further leveraged to reduce the volume of coherence traffic, and thus the coherence load on the shared memory modules and the interconnection network. If coherence on a dirty cache block is delayed, fewer coherence operations are necessary because multiple updates to the same cache block can be merged and made coherent in one coherence operation. The coherence volume reductions due to delaying coherence can be significant for software DSMs because they employ a large coherence unit, a page. The larger the coherence unit, the greater the likelihood that multiple shared memory writes will fall on the same cache block and thus become merged.

Software DSMs that support an RC memory consistency model can maximize the number of updates that are merged by delaying the coherence on a dirty page for as long as possible–until a release operation is encountered. In such delayed update systems, no coherence operations are invoked on normal shared memory accesses. All updates occur locally and are buffered in the page cache in each node's physical memory. A data structure, known as the *Delayed Update Queue* [13], records all the pages in the page cache that are dirty. At a release operation, the processor performing the release is stalled and coherence is initiated on all the pages listed in the Delayed Update Queue. The stalled processor resumes computation only after its updates have been consolidated.

Multiple-Writer Protocols

The delayed update technique described above can be effective only if multiple writers are permitted simultaneously on the same page. Otherwise, delayed updates can only be applied to pages with a single writer and zero or more readers. The multiple-writer case is important because of *false sharing*. False sharing occurs when two shared memory accesses with distinct addresses fall on the same cache block [21]. The coherence protocol is fooled into believing that the accesses conflict because it treats each cache block as an indivisible unit of data. Therefore, communication results to maintain coherence even though the coherence is unnecessary. False sharing becomes more severe as the size of the cache block increases. Since the coherence unit in software DSMs is a page, false sharing can be quite severe.

Handling multiple writers requires the ability to merge the updates performed on two or more cache blocks into a single coherent copy of the cache block at the home node. With multiple writers, it is not sufficient to simply store the entire contents of a dirty cache block at the home node as is done in hardware DSMs since all but the last store would overwrite the updates of the other cache blocks. Instead, it is necessary to determine what locations in each cache block have changed and to only update those locations at the home node.

Protocols that support multiple writers perform *twinning* and *diffing* to enable the merge of multiple dirty cache blocks [13]. Any processor that wishes to write a page in its page cache must first make a copy of the page, known as a twin. All writes are performed on the original, leaving the twin unmodified. When it becomes necessary to enforce coherence on the page, a comparison is performed between the dirty page and its twin, producing an update list, or diff, that represents all changes made to the page since the creation of the twin. The diff is sent to the home node so that the updates can be merged into the home node's copy of the page. Diffs have the property that multiple diffs can be merged by simply appending them to form a single diff representing all the changes made collectively by the processors that modified the page.

Chapter 3

The Multigrain Approach

This chapter presents a novel approach to building large-scale distributed shared memory machines that couples several small- to medium-scale multiprocessors using page-based software DSM techniques. We call this the "multigrain approach" because a single transparent shared memory layer is synthesized through the cooperation of both fine-grain and coarse-grain shared memory mechanisms.

The multigrain approach stems from the tension between supporting efficient shared memory mechanisms for high performance and supporting low-cost design for cost-effectiveness in the construction of large-scale shared memory machines. Fine-grain shared memory mechanisms provide high performance on a broad range of applications. While recent research efforts have enabled such mechanisms to scale in terms of performance, large-scale fine-grain architectures have failed to exhibit cost efficiency because of the engineering challenges presented by maintaining tight coupling across a large number of processors. Conversely, loosely-coupled software DSMs that support coarse-grain shared memory mechanisms are cost-effective because they leverage commodity technology. However, because loose coupling comes at the expense of highly efficient shared memory mechanisms, coarse-grain architectures cannot provide scalable performance, particularly on applications that exhibit fine-grain sharing.

The multigrain approach mediates the tension between performance and cost by building shared memory machines hierarchically and leveraging both fine-grain and coarse-grain shared memory mechanisms at different levels in the hierarchy. Multigrain architectures support fine-grain mechanisms within small- to medium-scale multiprocessor nodes. The size of each node is scaled as much as possible for performance, but only to the extent that the cost incurred by scaling fine-grain mechanisms remains reasonable. As soon as nodes become prohibitive from a cost standpoint, further scaling is facilitated by coupling multiple nodes using coarse-grain mechanisms.

The rest of this chapter motivates and presents the multigrain approach in greater detail, providing the foundation for the work presented in the rest of the thesis. Since granularity plays an important role in determining both performance and cost in shared memory machines, we begin by defining shared memory granularity in Section 3.1. In Section 3.2, we examine how existing conventional shared memory architectures support granularity, and expose the tension between performance and cost. Finally, in Section 3.3, we introduce the multigrain approach.

3.1 A Definition of Grain

Granularity is a familiar term in computer systems research used to describe many different aspects of computation. For instance, *granularity of parallelism* refers to the size of threads in a parallel computation. Threads that run for only a small number of cycles before exiting are "fine-grained," while threads that run for a large number of cycles are "coarse-grained." Granularity has also been used to describe synchronization. An application can exhibit either fine-grained or coarse-grained synchronization behavior depending on whether processors within the application synchronize frequently or infrequently, respectively.

The sharing patterns exhibited by a shared memory application have been characterized using the notion of granularity as well. Qualitatively, fine-grained sharing implies frequent communication between processors of small units of data, while coarse-grained sharing implies infrequent inter-processor communication of large units of data. However, compared to other uses of granularity, the use of granularity to describe application-level sharing patterns is imprecise because it lacks a quantitative definition. To help illustrate this point, consider our two previous examples, granularity of parallelism and granularity of synchronization. The granularity of parallelism can be quantified in terms of the number of cycles a thread executes before terminating. The granularity of synchronization can be quantified in terms of the frequency with which processors synchronize. Even though sharing granularity is common terminology within the domain of shared memory machines, surprisingly, no precise definition exists. Furthermore, because grain has been used in the context of both space and time, we need a definition that integrates both space and time. In this section, we develop a definition for sharing granularity that has a quantitative foundation to provide precise terminology for the remainder of the thesis.

We define the sharing granularity exhibited by two shared memory accesses as fine-grain if they occur close in time and reference memory locations separated by a small distance. Otherwise, the two shared memory accesses are defined to exhibit coarse-grain sharing. To illustrate this definition, let us plot the shared memory accesses made by all processors on a two-dimensional plot with time on one axis and memory address on the other, as shown in Figure 3.1. A *grain vector* can be defined in this 2-space for any pairing of shared memory accesses performed by different processors whose direction and magnitude are determined by two components, one for space and one for time. *Spatial distance* is simply the number of distinct memory locations that separate the two shared memory accesses. A smaller spatial distance arises if the two memory accesses are destined to locations that are close together in physical memory. Conversely, a larger spatial distance arises if the two memory accesses are destined to locations that are far apart in memory. *Temporal distance* can be defined by the number of clock cycles between the issue of the two memory accesses by their respective processors. A small number of

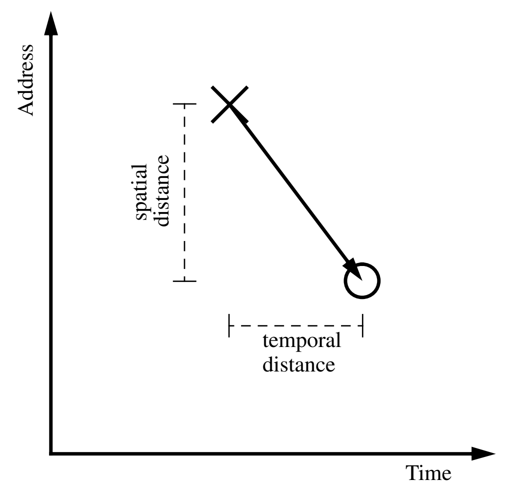


Figure 3.1: Defining sharing granularity using the *grain vector*. Two shared memory accesses, one made by a processor labeled "X" and another made by a different processor labeled "O," are plotted in space and time. Spatial and temporal distance components of the grain vector are labeled.

clock cycles indicates that the memory accesses are performed close in time, while a large number of clock cycles indicates that the memory accesses are performed far apart in time.

The two shared memory accesses in Figure 3.1 constitute sharing between two processors. The granularity of this sharing can be defined by considering both spatial and temporal distance components of the grain vector. Fine-grain sharing occurs when *both* spatial and temporal distance are small. Coarse-grain sharing occurs when *at least one* of the two components of distance is large. The requirement that both spatial and temporal distance are small for fine-grain sharing to occur is somewhat non-intuitive. Sharing granularity measures the degree to which shared memory accesses performed by different processors conflict. The definition above states that memory accesses conflict only if they occur close together in space and time. A real-world analogy is two friends who would like to meet face-to-face. For the meeting to occur, they must choose both a time and a place to meet. If either of these criteria are not met, the meeting will not occur.

Our definition of sharing granularity is illustrated visually in Figure 3.2. In the figure, spatial and temporal distance are specified along the X- and Y-axes, respectively, defining a plane. Each point inside the plane corresponds to a particular magnitude for a grain vector. The space of grain vector magnitudes is partitioned into those that represent fine-grain sharing, and those that represent coarse-grain sharing.

Figure 3.2 represents a simple definition of sharing granularity which is sufficient for the purposes of this thesis. Several issues must be further addressed in order to provide a more rigorous definition. We will discuss these issues only briefly here. First, we have drawn the partition between fine-grain and coarse-grain regions in Figure 3.2 in a circular fashion for simplicity, implying that sharing granularity is a function of

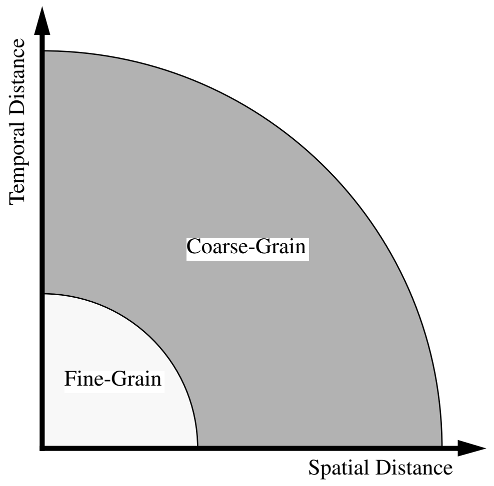


Figure 3.2: Classification of sharing granularity in terms of spatial and temporal distances. Fine-grain sharing occurs only if both temporal and spatial distance is small. All other sharing is coarse-grained.

only one variable, the grain vector magnitude. It may be desirable to formulate sharing granularity as a function of both spatial and temporal distance independently (leading to a partition boundary that is rectangular), or to adopt more complex formulations that take into account the direction as well as the magnitude of the grain vector. Second, Figure 3.2 shows a discontinuous transition between fine-grain and coarse-grain regions, implying that there is an abrupt delineation. In actuality, the transition should be continuous; therefore, rather than being simply fine-grained or coarse-grained, sharing would be characterized by some continuous quantity, for example, the magnitude of the grain vector. Finally, the development of our definition centers on two shared memory accesses performed by two processors. For the definition to be useful in describing the sharing behavior of an entire application, it is necessary to extend this definition to account for multiple accesses performed by multiple processors.

3.2 Granularity in Conventional Architectures

The presence of either fine-grain or coarse-grain sharing, as defined in the previous section, dictates the support needed by an application from a shared memory architecture in order to achieve high performance. In this section, we examine how conventional architectures support the demands of fine-grain and coarse-grain applications, and we discuss the implications of providing such support on system scalability and cost.

3.2.1 Supporting Fine-Grain Sharing

Applications that exhibit fine-grain sharing require aggressive architectural support in order to achieve high performance. As indicated by the definition of sharing granularity in Section 3.1, fine-grain sharing involves frequent conflicts between the shared memory accesses performed on different processors. Such conflicts invoke service from the shared memory layer in order to provide coherence on the shared data[1]. Because of the high frequency of conflicts (due to the small temporal distances separating conflicting accesses), significant shared memory overhead can be suffered leading to poor application performance unless the underlying architecture provides support for fine-grain sharing.

Hardware cache-coherent multiprocessors provide the architectural support necessary to efficiently support fine-grain sharing. Two architectural features are particularly important. First, these architectures support shared memory using special-purpose hardware. Consequently, the shared memory mechanisms are cheap, offering low-latency communication to applications. Low-latency shared memory mechanisms are crucial for fine-grain applications due to the high frequency with which the shared memory mechanisms are used in fine-grain sharing scenarios. Second, hardware cache-coherent architectures support sharing at the granularity of a cache line. Because the cache line is a relatively small unit of data, cache-coherent machines minimize the effects of false sharing conflicts[2]. False sharing is an important consideration for fine-grain applications since the small spatial separation between shared memory accesses characteristic of fine-grain sharing tend to induce false sharing conflicts.

While cache-coherent machines provide high performance on difficult fine-grain applications, the aggressive nature of the architectural support needed to efficiently handle fine-grain sharing poses several design challenges. In particular, it has proven difficult for cache-coherent architectures to simultaneously address two important design goals in the context of large-scale multiprocessors: scalability and cost-effectiveness. To date, the two cache-coherent architectures that have survived the test of time, Symmetric Multiprocessors (SMPs) and Distributed Shared memory Multiprocessors (DSMs), have only managed to each address these design goals separately.

The SMP is a popular cache-coherent architecture in which physical memory is centralized. Tight coupling is achieved by connecting processors via a bus. Each processor is responsible for maintaining coherence on data in its own hardware cache by *snooping* shared memory transactions that are broadcasted across the bus interconnect [20]. As a fine-grain architecture, the SMP offers the advantage of simplicity, due in large part to the existence of a broadcast primitive made possible by the bus interconnect. Its simplicity has contributed to its success as a cost-effective architecture. However, the very architectural feature that contributes to its economic success, the bus, renders SMPs unscalable in performance. Because communication and physical memory resources are centralized, the number of processors can increase only as long as adequate bandwidth exists to support communication between processors and memory. Beyond a certain point of scaling, the bus and single memory interface becomes a bottleneck through which shared memory

[1]In this section, we deliberately avoid discussing the specific sources of overhead in shared memory systems because it is peripheral to the main point of the section. Sections 7.1.1 and 7.1.2 of Chapter 7 address this topic in much greater detail.

[2]False sharing can still be a problem at the cache-line level. The trend in modern processors is increasing cache-line size, so the problem will get worse in future generations of cache-coherent machines.

transactions serialize resulting in significant performance degradation.

Hardware DSMs address the scalability problems encountered in SMPs. In the hardware DSM, physical memory is distributed across separate processing nodes such that each node owns a portion of globally shared memory. Nodes are connected via an interconnection network, and communicate across the network via point-to-point message passing. Special-purpose shared memory hardware on each node services shared memory requests made by the local processor, either by satisfying requests locally, or through communication with a memory module on a remote node via messages. Cache coherence is maintained using directory-based cache-coherence protocols, such as the one in [3]. Because communication and memory resources are distributed, the available communications bandwidth provided between processors and physical memory scales with the number of nodes in the system since each node has its own dedicated communications and memory interfaces. The ability to scale communication and memory bandwidth alongside processor count allows hardware DSMs to provide scalable performance.

While DSMs provide scalable performance on fine-grain applications, they enjoy far less success as cost-effective scalable architectures. Several factors prevent DSM architectures from being cost effective. Power distribution, clock distribution, and cooling do not scale linearly in cost at large machine sizes. Furthermore, physical constraints due to the need to maintain tight coupling often demand special packaging which further increase cost. From an engineering standpoint, it is difficult to provide tight coupling across a large number of processors in a cost-efficient manner. In addition to these engineering constraints, and perhaps most importantly, economic forces also limit the cost-effectiveness of hardware DSM architectures because high volumes are hard to achieve on large machines. Since the mechanisms that make hardware DSMs scalable are provided in hardware, these architectures are "over-designed" for applications in which smaller-scale commodity systems are adequate. By their very nature, hardware DSMs target large-scale problems. Unfortunately, large-scale problems, while they are important, do not drive the market for multiprocessors; most of the volume belongs to commodity applications. The lack of high volume demand prevents hardware DSMs from enjoying the economy of high volume production[3].

3.2.2 Supporting Coarse-Grain Sharing

Coarse-grain applications impose far fewer demands on architectural support for high performance as compared with fine-grain applications. As indicated by the definition of sharing granularity in Section 3.1, coarse-grain sharing is characterized by large temporal and spatial distances, implying that shared memory accesses performed by different processors seldomly conflict. This significantly relaxes the need for highly efficient shared

[3]It is conceivable (though it has not been demonstrated to date) that DSM technology could be applied to medium-scale systems. Such systems could have wider applicability and thus achieve higher levels of production, possibly making them commodity items. Existing DSMs, however, do not possess this economic advantage.

memory mechanisms. In fact, for coarse-grain sharing, it is often feasible to support shared memory purely in software without impacting performance. Furthermore, because of the larger spatial distances separating shared memory accesses, a coarser unit of sharing can be used without introducing significant false sharing conflicts. For some applications, it is even more beneficial to support a larger unit of sharing due to the amortization of protocol processing overhead per transaction across more data.

Page-based software distributed shared memory architectures are designed to capitalize on the less stringent architectural demands of coarse-grain applications. Like hardware DSMs, software DSMs have distributed communications and memory resources. However, software DSMs do not rely on any special-purpose hardware; instead, they leverage commodity technology throughout their implementation. Software DSMs use commodity uniprocessor workstations as processing nodes, commodity networks and network interfaces as the communications fabric, and they support shared memory mechanisms purely in software. Caching is performed in main memory using the page as the unit of sharing, while coherence on cached data is maintained using the same directory-based techniques employed in hardware DSMs implemented purely in software.

Because they leverage commodity technology, software DSMs are extremely cost-effective architectures. The key to their cost-effectiveness is two-fold. First, the components from which they are constructed are high volume components, and therefore, are themselves cost-efficient. Second, building large systems do not require any special costs, as was the case in hardware DSMs that require tight coupling. In fact, the original motivation for the software DSM architecture was to use the workstations and networks that already exist in a local area environment as the target machine. In this case, scaling up to large configurations incurs zero cost since the hardware already exists.

Unfortunately, the lack of architectural support for efficient shared memory mechanisms precludes software DSMs from delivering high performance on fine-grained applications. Remote memory accesses that use software to page across a commodity network are simply too costly to support the high frequency use of shared memory mechanisms characteristic of fine-grain sharing. Furthermore, the large coherence unit, a page, results in significant false sharing conflicts adding to the high frequency of shared memory activity. While much research has been devoted to minimizing the frequency of communication through protocol optimizations, it remains impossible for software DSMs to handle fine-grain sharing efficiently.

3.3 Multigrain Systems

Section 3.2 uncovered several design challenges that architects of large-scale shared memory machines face. These challenges can be summarized as follows.

1. Supporting fine-grain applications requires aggressive architectural support to provide tightly-coupled highly-efficient shared memory mechanisms.

2. Conventional architectures that support fine-grain applications efficiently have been

unable to achieve both scalable performance and cost-effective design. While hardware DSMs support fine-grain sharing in a scalable fashion, they are not cost-effective architectures due to the difficulty in supporting tight coupling for large-scale systems in a cost-efficient manner, and specialization to high-performance low-volume applications.

3. Architectures that leverage commodity technology are highly cost-effective. By implementing shared memory mechanisms in software, large-scale machines can be constructed from commodity components. However, these architectures cannot support fine-grain applications because software shared memory mechanisms lack necessary efficiency.

Our survey of conventional architectures exposes a tension between two seemingly conflicting goals: supporting fine-grain sharing in a fashion that provides scalable performance, and leveraging commodity technology for cost-effective designs. Furthermore, existing architectures are positioned at two extremes across the spectrum of cost and performance. Hardware DSMs are positioned at the high-performance extreme, and software DSMs are positioned at the low-cost extreme. The ability to trade off both cost and performance to explore intermediate points along the cost-performance spectrum do not currently exist.

The lack of "intermediate architectures" between hardware DSMs and software DSMs is one of the motivations for the multigrain approach. In this section, we describe multigrain architectures at an introductory level. Chapter 4 continues the discussion in greater depth where the design of a multigrain shared memory system, called MGS, is presented.

3.3.1 A Hybrid Approach for Both Cost and Performance

One of the reasons for the poor cost-effectiveness of large-scale hardware DSMs is the insistence on providing support for fine-grain sharing across the entire machine. The global nature of the fine-grain mechanisms in large-scale hardware DSMs necessitates tight coupling between all nodes in the system. As system size increases, tight coupling on a machine-wide scale becomes costly. However, eliminating tight coupling altogether is prohibitive since it would sacrifice the ability to support fine-grain applications.

The solution we propose in this thesis is to provide *some* tight coupling, but not across the entire machine. "Neighborhoods" of tight coupling can be formed by using special-purpose hardware to support cache-coherent shared memory within small- to medium-scale multiprocessor nodes. Tight coupling is necessary for performance, but the amount provided should only be to an extent that remains cost effective. Shared memory between cache-coherent nodes is supported via page-based software DSM techniques. Therefore, a single transparent shared memory layer is synthesized through the cooperation of both fine-grain and coarse-grain shared memory mechanisms, hence the name *multigrain shared memory*. The multigrain approach, as this is called, represents an intermediate solution compared against hardware DSMs which provide tight coupling across the entire machine and software DSMs which provide no tight coupling at all.

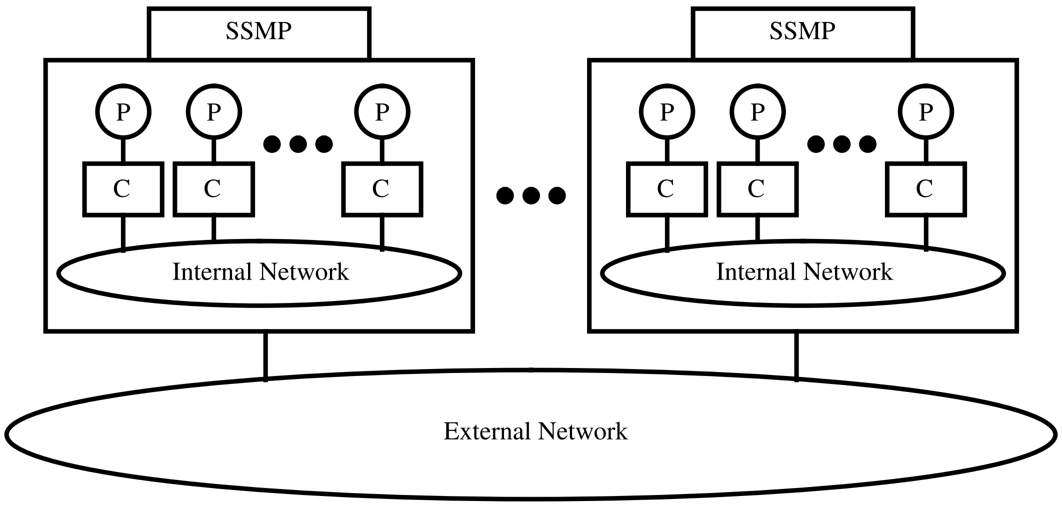


Figure 3.3: A Distributed Scalable Shared Memory Multiprocessor (DSSMP).

Multigrain systems have the potential to achieve both high performance and low cost. The existence of hardware support allows fine-grain sharing to be supported efficiently inside a multiprocessor node. Although only coarse-grain sharing can be supported by the software shared memory between nodes, multigrain systems still offer higher performance on fine-grain applications than software DSMs since *some* fine-grain mechanisms are provided. The extent to which multigrain systems can effectively support arbitrary fine-grain applications is a central topic of the evaluation of multigrain shared memory systems presented in Chapter 6. Multigrain systems are also much more cost-effective than hardware DSMs. Even though they require hardware support for shared memory, multigrain systems incorporate hardware support only on a small- or medium-scale. The amount of hardware support needed for good performance will also be a primary topic of Chapter 6.

3.3.2 DSSMPs

In this thesis, we refer to multigrain shared memory architectures as **D**istributed **S**calable **S**hared memory **M**ulti**P**rocessors (DSSMPs). Figure 3.3 shows the major components in a DSSMP. A DSSMP is a distributed shared memory machine in which each DSM node is itself a multiprocessor. In keeping with the terminology, these nodes are called **S**calable **S**hared memory **M**ulti**P**rocessors (SSMPs), as illustrated in Figure 3.3. *SSMP* is an architecture-independent name for any small- to medium-scale cache-coherent shared memory machine. We envision two candidate architectures for the SSMP: the bus-based Symmetric Multiprocessor (SMP), or the small- to medium-scale NUMA multiprocessor[4].

[4]Architecturally, a small- to medium-scale NUMA multiprocessor resembles the large-scale hardware DSM, but is targeted for smaller systems. An example of such a system might be the SGI Origin [46] in a small-scale configuration.

As Figure 3.3 shows, DSSMPs have two types of networks that form the communication substrate: an internal network and an external network. The internal network provides interconnection between processors within each SSMP. In the case that the SSMP is a symmetric multiprocessor, this network is a bus[5]. In the case that the SSMP is a NUMA multiprocessor, it may be a switched point-to-point network. The external network connects the individual SSMPs and consists of a high-performance local area network (LAN), such as ATM or switched Ethernet.

In addition to a hierarchy of networks, DSSMPs also provide shared memory support in a hierarchical fashion. Each SSMP provides special-purpose hardware for cache-coherent shared memory. This may take the form of snoopy-based cache coherence in the case of SMPs, or directory-based cache coherence in the case of NUMA multiprocessors. Between SSMPs, shared memory is supported using page-based software shared memory.

Chapter 4 discusses the design of multigrain shared memory in detail. In the rest of this section, we provide an introductory explanation of some important design principles to give a "flavor" for how multigrain systems are constructed. Figure 3.4 shows an example of how data is distributed across processors in a DSSMP to illustrate the key computation structures required in any multigrain shared memory system. The example shows four different processors on three separate SSMPs that access data in the same page. In order for the processors to access the data, the page containing the data must first be replicated in the physical memory of their respective SSMPs. Figure 3.4 shows three replicated pages, on SSMP0, SSMP1, and SSMP2, respectively. Notice that each page can have either read-only or read-write privilege depending on the type of accesses made by the processor(s) on the SSMP.

Once a copy of the desired page resides in the SSMP's physical memory, any processor on the SSMP can access the data in the page. A mapping entry in the processor's TLB provides the address translation that allows the processor to name locations in the page. Like the access privilege on the page itself, this mapping entry can allow either reads or reads and writes on the page. Finally, hardware cache-coherent shared memory further replicates the data into the hardware caches that deliver the data to the processors. In Figure 3.4, one processor on SSMP0, a second processor on SSMP1, and a third processor on SSMP2 are reading data from the page; therefore, all three processors have a read mapping of the page in their TLBs, and read copies of the data in their hardware caches. At the same time, one processor on SSMP2 is performing reads and writes on the page. This processor has a read-write mapping in its TLB and read-write copies of the data in its hardware cache.

The key point in Figure 3.4 is that data distribution in a DSSMP happens hierarchically. First, data is replicated across SSMPs in units of pages via software shared memory, resulting in a copy of the page being placed in the physical memory of the SSMP. Once a copy of the page resides in the SSMP, the data is further replicated to individual (and potentially multiple) processors in the SSMP in units of cache lines via

[5]To enhance scalability, some SMPs are moving away from buses and are using switched interconnect. An example of this is the SUN Enterprise Server [65].

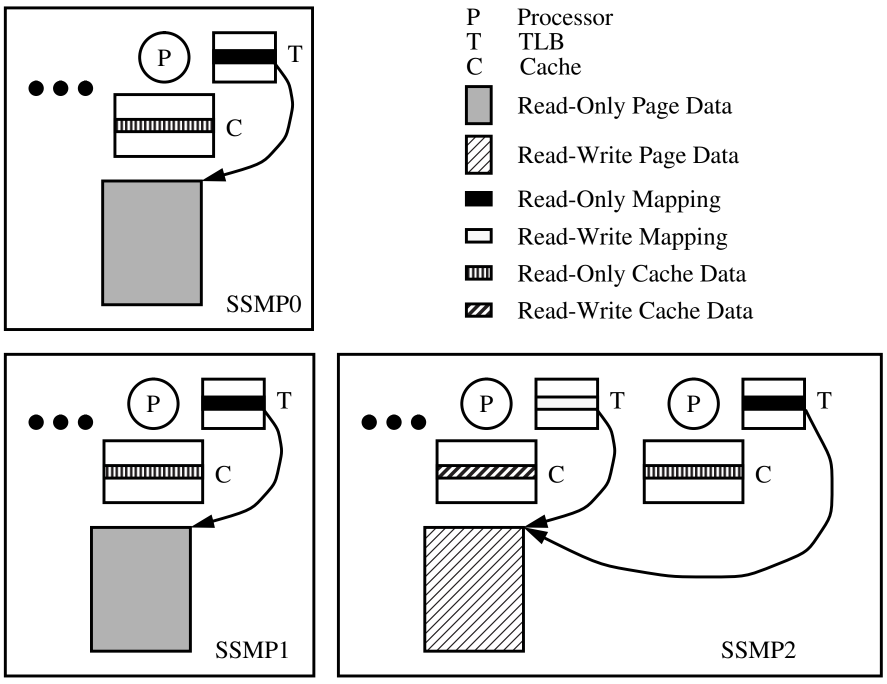


Figure 3.4: Distribution of a single page of data across three SSMPs. Only the processors involved in sharing are shown; there are other processors in each SSMP that are not shown.

hardware cache-coherent shared memory.

The hierarchical data distribution scheme illustrated in Figure 3.4 reflects a variation, both in spatial grain and temporal grain, as shared memory accesses traverse different levels in the memory hierarchy.

Spatial grain. The coherence unit size changes as shared memory accesses cross SSMP boundaries. Within an SSMP, the coherence unit size is a cache line since data distribution and replication is supported by cache-coherence hardware. Between SSMPs, the coherence unit size is a page since shared memory is supported by page-based software shared memory.

Temporal grain. Latency of shared memory accesses increases significantly as SSMP boundaries are crossed for two reasons. First, shared memory transactions experience drastically different latencies depending upon whether they are supported in hardware or software. Second, communication across the internal network is much cheaper as compared to the external network. The internal network provides raw hardware interfaces directly to the hardware shared memory mechanisms. Typically, the external network is unreliable and untrusted. Building reliability and security over the external network requires running expensive protocol stacks in software.

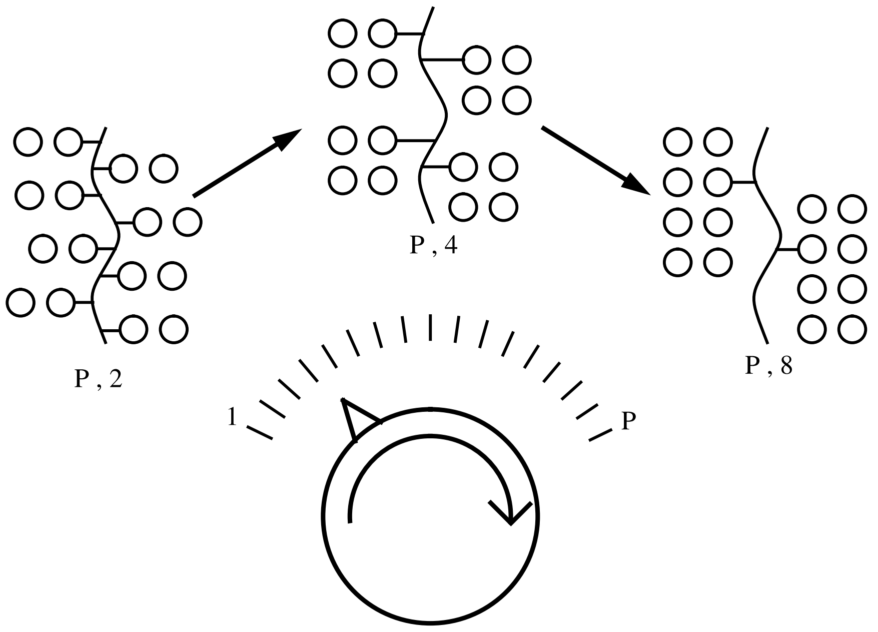


Figure 3.5: A family of DSSMPs is defined by fixing the total processing and memory resources, and varying the SSMP node size. The notation P, C denotes a DSSMP with P total processors, and C processors per SSMP node.

The variation of spatial and temporal grain at SSMP boundaries impacts the granularity of sharing experienced by different processors in the DSSMP. Shared memory accesses performed by processors colocated on the same SSMP experience fine-grain sharing due to the small spatial and temporal grain supported by cache-coherence hardware. Shared memory accesses performed by processors on separate SSMPs experience coarse-grain sharing due to less efficient page-based software shared memory.

3.3.3 DSSMP Families

As mentioned earlier in the section, DSSMPs represent the "intermediate architecture" along a cost-performance spectrum whose endpoints are the hardware DSM and the software DSM. Like any other spectrum in nature, the spectrum of shared memory architectures that we have defined based on sharing granularity is somewhat continuous. Before we embark on the study of machines along this spectrum which is the objective of this thesis, it is beneficial to more precisely characterize the spectrum.

While there are many system parameters that characterize the configuration of a parallel machine, a key parameter is the system size, or the number of processing elements in the system, P. DSSMPs can also be characterized in this fashion; however, another key parameter in the case of DSSMPs is the SSMP node size, C. Therefore, we can introduce the notation P, C to crisply identify specific configurations of DSSMPs.

Many DSSMP configurations are similar; in particular, we say that all configurations with the same P parameter belong to the same *DSSMP family*. As illustrated in Figure 3.5, a family of DSSMPs is defined by fixing the total number of processing elements, P, and varying SSMP node size[6]. DSSMPs in the same family differ only in the way processors are clustered.

This taxonomy of DSSMPs turns out to be useful for characterizing the spectrum of machines discussed above for the following reason. The clustering boundary, *i.e.* the boundary that divides processors on the same SSMP from those that are on remote SSMPS, determines where hardware-supported shared memory meets software-supported shared memory. Therefore, by varying SSMP node size, we in effect vary the mix of fine-grain and coarse-grain support for sharing between processors. DSSMPs with smaller SSMP nodes rely more on software-supported shared memory and provide more coarse-grain sharing support. Conversely, DSSMPs with larger SSMP nodes rely more on hardware-supported shared memory and provide more fine-grain sharing support. Furthermore, the conventional shared memory machines described in Section 3.2 are captured by our taxonomy as degenerate configurations at the endpoints of the spectrum. All-software DSMs are the $P, 1$ configurations, while all-hardware DSMs are the P, P configurations.

The most important aspect of our taxonomy is that it points to the existence of a "knob," as depicted in Figure 3.5. This knob is not only a SSMP node size knob and a sharing granularity knob, but it also serves as a knob for tuning cost against performance. The knob illustrates how multigrain systems are in fact an answer to the plea for an intermediate architecture that was posed at the beginning of this section. The importance of Figure 3.5 is a theme that will reappear in future parts of the thesis.

[6] In this thesis, we only consider SSMP node sizes, C, that divide P evenly. Otherwise, the DSSMP will contain SSMPs of varying sizes, in which case a single SSMP node size parameter cannot specify the sizes of all SSMPs in the system.

Chapter 4

MGS System Architecture

This chapter proposes a system architecture for multigrain shared memory, called *MGS*[1]. The discussion of the architecture proceeds in four parts. The first two parts, Sections 4.1 and 4.2, present our design of multigrain shared memory. Section 4.1 describes the MGS mechanisms—these include mechanisms found in existing hardware and software DSM systems, as well as novel mechanisms that are needed specifically for multigrain systems. Section 4.2 describes the structure of the MGS architecture, focusing on the major architectural pieces, and how the pieces interact. Our intention for these two design sections is to provide a clean and lucid exposition of the most important aspects of the design. We do not intend to provide exhaustive blueprints for the entire system. The interested reader is encouraged to study Appendix A where a complete and detailed specification of MGS is given.

The third part, Section 4.3, presents a user-level synchronization library that accelerates synchronization operations on clustered systems by leveraging the fast communication mechanisms within SSMPs whenever possible. Finally, Section 4.4 presents the programmer's interface exported by MGS.

4.1 Enabling Mechanisms

MGS couples hardware cache-coherent shared memory with software page-based shared memory. The hardware layer provides shared memory within SSMPs, while the software layer extends hardware-supported shared memory across SSMPs in as seamless a fashion as possible. Figure 4.1 presents a schematic of the MGS system, illustrating the layered construction of the system by representing each major system layer with a box. Boxes drawn with solid lines represent layers of the system that are implemented in hardware, and boxes drawn with dotted lines represent layers of the system that are implemented in software. As Figure 4.1 shows, there are three major system layers in the construction of multigrain shared memory. Two major pieces are the hardware cache-coherent and

[1]The name *MGS* derives from the acronym for **M**ulti**G**rain **S**hared memory. It refers to the particular multigrain shared memory system proposed in this thesis.

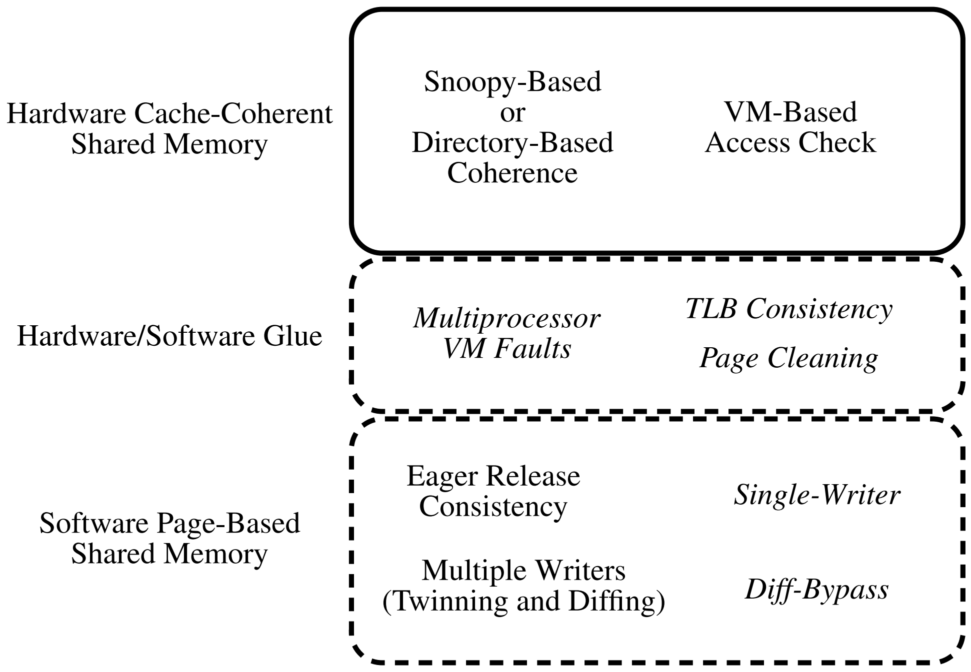


Figure 4.1: Schematic of the MGS system. Italicized text reflect new mechanisms needed for multigrain shared memory.

software page-based shared memory layers already mentioned. In addition, an intermediate software layer provides the glue or interface between the traditional hardware and software shared memory layers.

Each of the three layers in MGS implements several key mechanisms that together support multigrain shared memory. Figure 4.1 shows these mechanisms inside the layers to which they belong. Because multigrain shared memory is fundamentally the combination of cache-coherent and page-based shared memory, many of the mechanisms proposed for conventional distributed shared memory discussed in Chapter 2 are also needed in multigrain systems. These mechanisms are printed in normal text in Figure 4.1. However, there are also several additional mechanisms that are needed specifically for multigrain shared memory. These mechanisms are printed in italicized text in the figure.

The goal of this section is to describe all the mechanisms in Figure 4.1. We begin by briefly describing the mechanisms borrowed from conventional shared memory systems. Then we describe the new mechanisms needed specifically for multigrain shared memory.

4.1.1 Conventional Shared Memory Mechanisms

MGS relies on two hardware mechanisms. The first is hardware cache-coherent shared memory to provide the shared memory abstraction within SSMPs[2]. We envision two pos-

[2] What we have lumped into a single mechanism here in fact is a potentially very complex system involving many hardware mechanisms. More details of hardware cache-coherent shared memory can be

sible hardware cache-coherent shared memory architectures for the SSMP: the bus-based symmetric multiprocessor, and the small- to medium-scale distributed shared memory multiprocessor. In this thesis, MGS has been designed assuming the DSM architecture; however, our approach can be applied to the symmetric multiprocessor architecture as well with some modifications. The specific implementation of cache-coherent shared memory within SSMPs is not crucial; what is important is that the hardware supports a shared memory address space, and that it provides the mechanisms for maintaining coherence on cached shared data.

Second, MGS relies on hardware support for virtual memory, namely support for address translation and protection checking as is commonly provided by hardware TLBs[3]. Virtual memory is a requirement because software shared memory provides a shared virtual address space in the absence of shared physical memory. Address translation provides the level of indirection between shared virtual memory and local physical memory. Protection checking is the mechanism by which accesses to shared virtual memory get trapped if they have no backing in physical memory. Once trapped, an access is forwarded to the software layers that provide shared memory mechanisms between SSMPs. (See Chapter 2 for more details).

While the mechanisms provided by the hardware layer are an integral part of the overall MGS system, we are less concerned with their design as compared to the design of the software layers for two reasons. First, one goal of the thesis is to target commodity hardware technology. We are interested in leveraging hardware platforms that can be found commercially. This limits our ability to dictate the design of the hardware mechanisms. Second, and in some ways a more compelling consideration, the specific design of the hardware layer is less crucial to the behavior of the overall system because the software is usually the bottleneck. To first order, it doesn't matter how we design the hardware mechanisms (as long as they support the conventional mechanisms described above) because the hardware will be fast compared to the software. For these reasons, we will not discuss the hardware mechanisms further in this section.

The software layer in MGS that supports page-based shared memory borrows many mechanisms from conventional software DSM systems that run on clusters of uniprocessor workstations. Our design of software page-based shared memory is closest to the Munin system [13]. In particular, we support a release consistent (RC) memory consistency model using the Delayed Update Queue structure proposed by Munin (see Section 4.2.3). In addition, we support multiple writers via *twinning* and *diffing*, another technique first proposed by Munin. Together, these software shared memory mechanisms significantly alleviate coherence overhead introduced by false sharing. (See Chapter 2 for more discussion on these mechanisms).

---

found in Chapter 2.

[3]As we will discuss in Section 5.3.1 of Chapter 5, our implementation of MGS uses a hardware platform that does not provide hardware support for virtual memory; instead, we support virtual memory in software. However, VM has traditionally been supported in hardware on most platforms; therefore, we consider this mechanism belongs to the hardware layer.

4.1.2 Additional Mechanisms for Multigrain Shared Memory

In addition to the mechanisms borrowed from conventional shared memory, there are novel mechanisms in Figure 4.1 that are needed specifically for multigrain shared memory (those in italics). We first describe the novel mechanisms in the glue layer, then we describe the novel mechanisms in the software DSM layer.

MGS provides three mechanisms that form the glue or interface between the hardware and software shared memory layers. The three glue mechanisms are *multiprocessor VM fault handlers*, *page cleaning* and *mapping consistency*[4]. Multiprocessor VM fault handlers allow the vectoring of VM faults from multiple sources within an SSMP node which is necessary since each node is a multiprocessor instead of a uniprocessor. The page cleaning mechanism bridges the gap in coherence between the hardware and software shared memory layers. And the mapping consistency mechanism provides coherence on mapping state within an SSMP.

Multiprocessor VM Faults

As we described above, the software layer in multigrain shared memory closely resembles software DSM in conventional systems. The differences, however, stem from the fact that each DSM node in MGS is a multiprocessor, not a uniprocessor. Consequently, MGS will encounter VM protection faults from multiple sources within each software DSM node. Furthermore, the action taken in response to a particular fault may be very different under MGS.

To illustrate these differences, Table 4.1 lists the basic fault types and the action taken on each fault in both conventional DSMs on uniprocessor workstations, and DSM in MGS. The table shows what happens when a processor on the DSM node makes either a read or write access, specified in the "Access" column, to a shared page. The page can either be mapped read-only or read-write in the processor's TLB, or not mapped at all ("Invalid" state), specified in the "TLB State" column. The page data can reside in the node's local memory with either read-only or read-write access privileges, specified in the "Page State" column. Pages that have not been paged into local memory are specified as "Invalid." The last two columns in Table 4.1 specify the fault that occurs given the access type and state of the TLB and page data, and the action taken by the software shared memory layer, respectively.

In conventional systems, the state of the TLB and the state of the page data, *i.e.* mapping and data, are synchronized, as illustrated by the "TLB State" and "Page State" columns in Table 4.1. Furthermore, every VM protection fault requires an action that

[4]Mapping consistency is similar to *TLB consistency*, a term more widely recognized. The reason we do not use the popular terminology is because our implementation of MGS on Alewife supports address translation in software, and thus does not have TLBs. Consistency on mapping state is still necessary, but software structures are involved instead of hardware TLBs. The problem is the same, but because our system does not have hardware TLBs, we choose the more generic term "mapping consistency." In this section, we will treat mapping consistency as a solution for a system with hardware TLBs, and defer discussion specific to our implementation until Chapter 5.

Access	TLB State	Page State	Fault Type	Action
Conventional				
Read	Invalid	Invalid	Page Fault	Page In
Write	Invalid	Invalid	Page Fault	Page In
Write	Read-Only	Read-Only	Page Fault	Page Upgrade
MGS				
Read	Invalid	Invalid	Page Fault	Page In
Write	Invalid	Invalid	Page Fault	Page In
Write	Read-Only	Read-Only	Page Fault	Page Upgrade
Read	Invalid	Read-Only	TLB Fault	TLB Fill
Write	Invalid	Read-Write	TLB Fault	TLB Fill

Table 4.1: A comparison of possible fault events encountered by the software DSM layer in a conventional software DSM system and in the MGS system.

touches page data, such as paging between DSM nodes ("Page In" action) or upgrading the access privilege on an existing page ("Page Upgrade" action)[5]. As we will discuss in Section 4.2, these actions are expensive because they require communication between DSM nodes.

In MGS, mapping and data state are not necessarily synchronized. Initially, when a processor faults on a page that is not resident in the local node's memory (first two rows under "MGS" in Table 4.1), or when a processor makes the first write access to a read-only page (3rd row under "MGS"), actions occur on the page data. These actions can benefit the other processors in the same SSMP since subsequent faults do not need to repeat the actions on the page data. Instead, they only need to manipulate mapping state, as indicated by the last two rows in Table 4.1 which show that some faults can be satisfied with a TLB fill operation. These faults are much less expensive than faults that touch page data since they can be completed locally on the SSMP.

Page Cleaning

The page cleaning mechanism maintains a single view of coherent data as seen by the hardware and software shared memory layers. Because of replication in hardware shared memory, the contents of a page in physical memory may not represent a coherent version of the page. For instance, there may be one or more cache lines in the page that are dirty in a processor's cache somewhere in the SSMP. If the software DSM protocol tries to move such a page (for instance, during an invalidation operation), it may see incoherent data.

The problem arises because movement of a page out of an SSMP occurs through a network interface. Such interfaces typically perform data transfer by using DMA that is

[5] In this discussion, we ignore the effects of TLB capacity. Because of replacements in the TLB, it is possible for a page to reside in a node's local memory, but no mapping to exist in the TLB. This can result in a TLB Fault in which a TLB fill occurs without any page data operations.

not coherent with respect to the processor caches. For the data transfer to see coherent data, all hardware-distributed copies need to be localized[6].

There are several different ways of localizing page data. MGS employs an all-software approach, called page cleaning. In page cleaning, the processor that initiates the localization operation walks the entire page. For each cache line in the page, the processor forces the cache-coherence hardware to issue an invalidation for the cache line. After this is completed for all cache lines in the page, we are guaranteed that the data from the page is purged from all the processor caches. Section 5.3.3 of Chapter 5 discusses page cleaning in more detail, including how to make it go fast.

Mapping Consistency

The last glue mechanism is mapping consistency. Since the software DSM protocol distributes pages across SSMPs and then reclaims them via invalidation, it must be able to access and modify the address mapping state within the individual SSMPs. A consistency problem arises because the mapping state, located in page tables, can be cached in the TLBs of potentially multiple processors. The system must ensure that any changes made to the mapping state in the page tables do not leave a stale copy of a mapping entry in any processor's TLB.

Many approaches for providing mapping consistency (also known as *TLB consistency*) have been proposed [17, 64, 54, 9]. The solution used in MGS is closest to the one proposed in the PLATINUM system [17]. Each SSMP has a TLB directory that tracks the cached page table entries for all the pages resident in the SSMP's physical memory. The TLB directory is updated whenever a TLB fill is performed by marking the ID of the processor caching the mapping entry. When a page table entry is modified, the TLB directory is consulted and an invalidation request for that entry is posted to all processors which have cached the entry in their TLB via inter-processor interrupts. Processors can be interrupted selectively because the TLB directory specifies the exact set of processors with the mapping cached in their TLBs. Without the directory, all processors would have to be interrupted thus resulting in much higher synchronization overhead.

Since it is possible for concurrent accesses to occur on page table state, MGS provides a lock for every page table entry to ensure atomic access. Processors that wish to read a page table entry during TLB fill or modify an entry during TLB invalidation must acquire the lock before performing the access. No attempt is made to distinguish between read accesses and write accesses (using readers and writers locks, for instance) since the frequency of accesses to the page table is low enough that serialization is not a significant problem. Section 4.2.2 gives further discussion on the locking scheme used for mapping

[6]There is another coherence problem that is symmetric to the invalidation case. Suppose a page is returned to the operating system's pool of free pages before all the data inside the page is localized. At a future point in time, the SSMP reallocates the page to receive data from a remote SSMP via DMA that is not coherent with processor caches. When this page is remapped, it is possible for processors to access stale data due to residual copies of the data in the hardware caches from the earlier mapping of the page.

consistency. Also, see Appendix A for specific details on how mapping consistency is implemented.

We now describe the two mechanisms in MGS implemented in the software shared memory layer. The first mechanism is called the *Single-Writer Mechanism* and is crucial for obtaining good performance on multigrain shared memory systems. The second mechanism is called the *Diff-Bypass Mechanism* and can be helpful for any shared memory protocol that supports multiple writers via diffing. Both mechanisms address performance. The correctness of MGS does not rely on these mechanisms, but the mechanisms significantly improve MGS performance (especially the Single-Writer mechanism).

Single-Writer

From the standpoint of performance, the goal of MGS is to maintain the illusion that the DSSMP performs as if it were a hardware cache-coherent shared memory machine in spite of the fact that software is used to support shared memory between SSMPs. To maintain this illusion as often as possible, MGS must leverage hardware-supported shared memory aggressively, particularly when sharing patterns permit the system to bypass software layers.

In general, it is difficult to avoid software intervention. Any shared memory operation that requires communication across SSMPs or that needs to modify mapping state within the same SSMP necessarily invokes software layers to provide the desired services. For instance, if a page is shared by processors on two or more SSMPs, then maintaining coherence on the page will require inter-SSMP communication and software protocol processing on each of the SSMPs[7].

One important sharing pattern that a DSSMP should handle efficiently, however, is when a page is shared only between processors colocated on the same SSMP. We call such a scenario a *Single-Writer condition.* The name "Single-Writer" reflects the fact that there is exactly one outstanding write copy of the page in the entire system, even though potentially multiple processors (in the same SSMP) are accessing the page. Sharing patterns that meet the Single-Writer condition should incur the minimum amount of software overhead: one page fault to bring the page into the SSMP by the first processor to access the page, and one TLB fault for every additional processor in the SSMP accessing the page to provide a mapping in that processor's TLB[8]. After this minimum software overhead is incurred, all shared memory accesses should be satisfied using hardware cache-coherent shared memory. Furthermore, no additional software overhead should be suffered until a processor on a remote SSMP wishes to access the page, thus violating the Single-Writer condition.

[7]Note that read sharing across SSMPs can be supported with no software intervention (aside from cold misses), but this is a trivial case because of the lack of a coherence problem. In this section, by sharing we mean that two or more processors perform accesses to the same page in which at least one processor is performing writes.

[8]There may be an additional fault to upgrade the page from read privilege to write privilege if the very first access made to the page was a read access, and subsequent accesses perform writes.

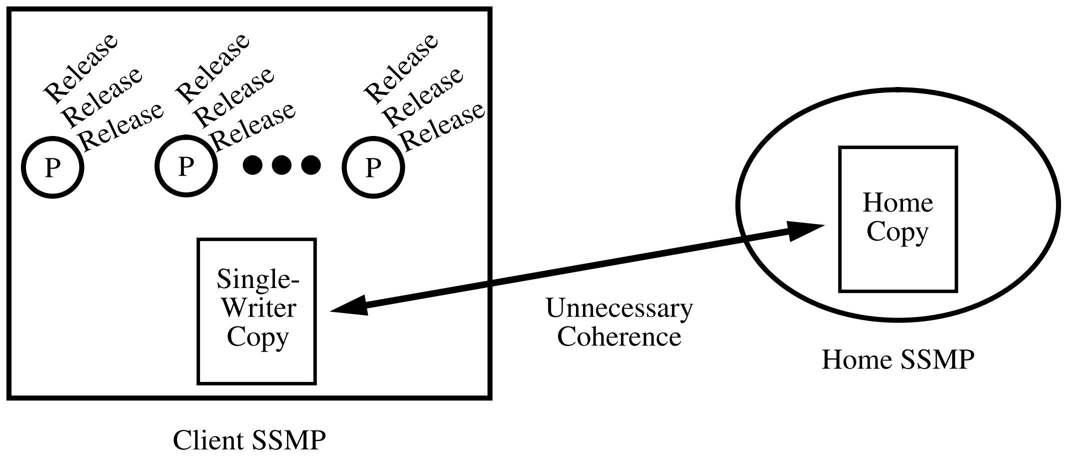


Figure 4.2: Unnecessary communication and protocol processing overhead for sharing under the Single-Writer condition.

The minimum software overhead for a page that meets the Single-Writer condition may not be achieved because of the explicit nature of coherence management in implementations of release consistency on software shared memory systems. For instance, software shared memory systems such as MGS that implement RC using delayed updates maintain coherence at every release (see Section 2.2.2 of Chapter 2 for a discussion on delayed updates). This coherence management policy prevents the efficient handling of sharing patterns that meet the Single-Writer condition. Figure 4.2 illustrates that each release under delayed updates requires communication to maintain coherence between at least two copies of a page, one at the client SSMP, and one at the page's home SSMP. For sharing patterns that obey the Single-Writer condition, this communication is unnecessary since the client is the sole SSMP accessing the data (*i.e.* under the Single-Writer condition, the home SSMP does not require updating at every release). Therefore, while the Single-Writer sharing pattern permits the copy at the home SSMP to remain incoherent past multiple client release operations, the strict adherence to release consistency prevents the elimination of this unnecessary communication.

One possible solution is to identify Single-Writer conditions statically in the source code, and then to transform the source code such that release operations are not emitted to the shared memory system. There are two problems with this approach. First, extra effort on the part of the programmer or compiler to identify shared memory accesses that obey the Single-Writer condition is needed. And second, in cases that exhibit dynamic behavior, it may not be possible to perform the transformation because meeting the Single-Writer condition cannot be guaranteed all the time. Under these circumstances, the programmer or compiler must be conservative and omit the transformation even if the Single-Writer condition can be met most of the time.

A better solution is to allow the shared memory protocol to identify the Single-Writer condition at runtime and for these cases, allow the protocol to relax coherence beyond the release point. This is what we call the *Single-Writer mechanism.* The mechanism has three parts: Single-Writer detection, relaxing coherence, and reverting to a normal level of coherence.

Single-Writer Detection. The Single-Writer condition is met when there is exactly one outstanding write copy of a page in the entire system. This condition can only be detected at a page's home SSMP where the page directory can be consulted (see Section 4.2.3). Each time a client SSMP performs a release and sends a request to the home SSMP for coherence, the home looks at the page's directory entry and determines whether the Single-Writer condition is met.

Relaxing Coherence. Normally, when the home SSMP receives a request for coherence from a client SSMP, it initiates invalidation on the page. For those pages that meet the Single-Writer condition as described under Single-Writer detection, the home SSMP instead sends a special message back to the client SSMP notifying the client that it should relax the coherence policy on this page. The client SSMP transitions its local copy of the page to a special *Single-Writer mode*. In this mode, all subsequent release operations performed by any processor in the SSMP are ignored by the software shared memory layer (the exact details concerning how this is accomplished are discussed in Section 4.2 and Appendix A). In essence, the system breaks the RC memory consistency model for that page because the system has detected that sharing patterns on the page do not require coherence. The home SSMP also marks the page's directory entry to indicate that the page has transitioned to the single-Writer mode.

Reverting to Normal Coherence. The system must revert to a normal coherence policy as soon as the Single-Writer condition is violated, which occurs when any processor on an SSMP other than the client with the Single-Writer copy tries to access the page (we will call this SSMP the "3rd-party SSMP"). When this happens, a page fault is guaranteed to occur on the 3rd-party SSMP since in the Single-Writer mode, there is only a single SSMP in the entire system with an outstanding copy. The page fault request from the 3rd-party SSMP will be received by the home SSMP which consults the page directory as usual. The home SSMP will then recognize the page is in the special Single-Writer mode. Before the home SSMP can service the page fault from the 3rd-party SSMP, it must first invalidate the Single-Writer copy. It initiates an invalidation and waits for an acknowledgment. When the invalidation completes, the contents of the Single-Writer copy will be returned to the home SSMP which is used to restore coherence on the home copy. At this point, normal coherence is restored and the 3rd-party SSMP page fault can be serviced.

Diff-Bypass

In a shared memory protocol that supports multiple writers, diffs are used to merge multiple dirty pages after an invalidation into a single coherent version of the page.

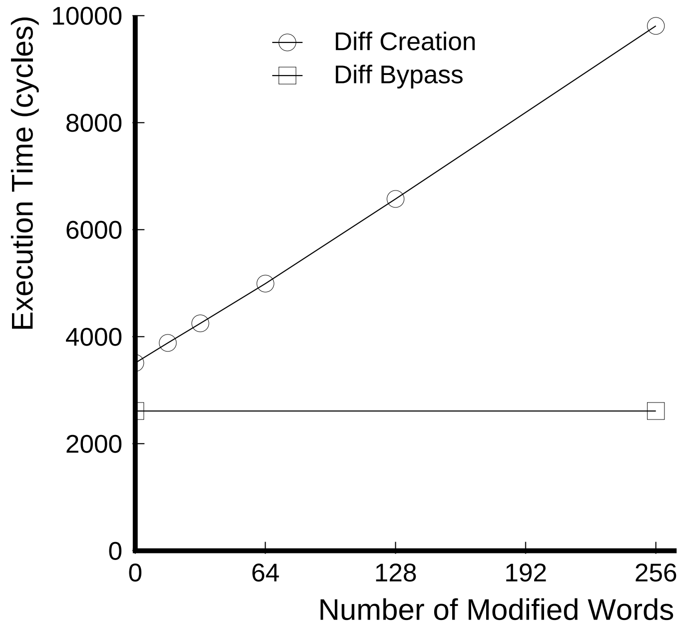


Figure 4.3: Overhead of Diff Creation and Diff Bypass as a function of the number of modified words in the page. Page size is 1K-bytes.

There are instances, however, where diffs are not necessary even when there are dirty pages outstanding during invalidation. For instance, if there is only a single dirty page outstanding, then the diff is not necessary. In fact, the dirty page itself represents the coherent copy of the page. Even when there are multiple dirty pages outstanding, it is still possible to avoid diff creation. One of the dirty pages can be selected to bypass diff creation. As long as diffs are created for all other dirty pages, the diffs can be merged into the selected page. Therefore, under these circumstances, the system has a choice between creating a diff or bypassing the diff.

From the standpoint of efficiency, choosing between diff creation or bypassing is a trade off between message size and computational overhead. Often, the size of the diff is smaller than the size of a page. For instance, in MGS, a diff contains two words for every modified word in the page (one word for the address, and one word for the new value). Therefore, if fewer than half the words in a page have been modified, then the diff will be smaller than the page. In this case, the cost of sending the diff will be smaller than the cost of sending the entire bypassed page. However, the reduction in data movement overhead must be traded off against the cost of computing the diff in the first place. The right choice depends on the number of modified words in the dirty page, and the specific architectural parameters of the target system.

Figure 4.3 shows a simple experiment performed on the Alewife multiprocessor in which the number of cycles required to compute a diff and send it to a neighboring processor is plotted against the number of modified words in the page, which is increased from 0 to the size of the page (in this example, the page size is 1 K-byte or 256 words). This curve is compared to the cost of sending a 1K-byte message directly without any computational overhead, which is the cost incurred by the Diff Bypass mechanism. Figure 4.3 shows that on Alewife, creation of a diff is always more costly than bypassing, even when there are no modified words in the page. The minimum cost of the "Diff Creation" curve is equal to the computational cost of performing 256 comparisons followed by sending a null message. This overhead is approximately 3500 cycles, which is more costly than sending a 1K-byte message on Alewife, an overhead of approximately 2600 cycles.

Figure 4.3 demonstrates that the right choice is to always bypass diff creation whenever possible. On Alewife, the network has sufficient bandwidth such that the computational overhead always dominates the data movement overhead. Of course, this may not be true for other systems; however, in this thesis, we assume that the networks used in a DSSMP will have high bandwidth. Therefore, in MGS, we always choose Diff Bypass when it is allowed.

MGS employs the Diff Bypass mechanism in the following manner. When an invalidation is initiated, the home SSMP decides whether diff creation is needed for each dirty page being invalidated. MGS provides different invalidation messages to signify whether a diff should be created on the client SSMP, or whether diff creation should be bypassed and the entire page should be sent back to the home SSMP. Diff-Bypass requires the home SSMP to wait for the bypassed dirty page to arrive before initiating the merge of any other diffs into the home copy of the page; otherwise, any premature merges would be lost[9].

4.2 Architectural Structure

This section discusses the structure of the MGS architecture. It highlights major architectural components, and explains how these components interact. We begin by identifying classes of shared memory operations, or transaction types, and describe the major structures involved in supporting these transactions (Section 4.2.1). Then, we look at how the correctness of the architecture can be compromised by simultaneous transactions, and we present several solutions that maintain correctness between transactions (Section 4.2.2). Finally, we discuss low-level components that are necessary to implement the architecture (Section 4.2.3).

4.2.1 Three-Machine Discipline

Distributed shared memory computers are constructed by implementing a number of distributed state machines which run a shared memory protocol. The protocol relies

[9]In fact, in MGS, the memory-side processor waits for *all* outstanding acknowledgments, including both diffs and bypassed dirty pages, to arrive before initiating merging of the diffs.

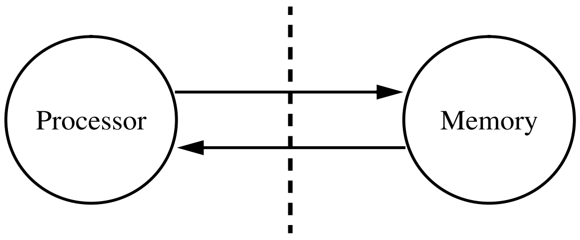


Figure 4.4: The 2-machine decomposition in conventional software DSM systems. The dotted line indicates a node boundary.

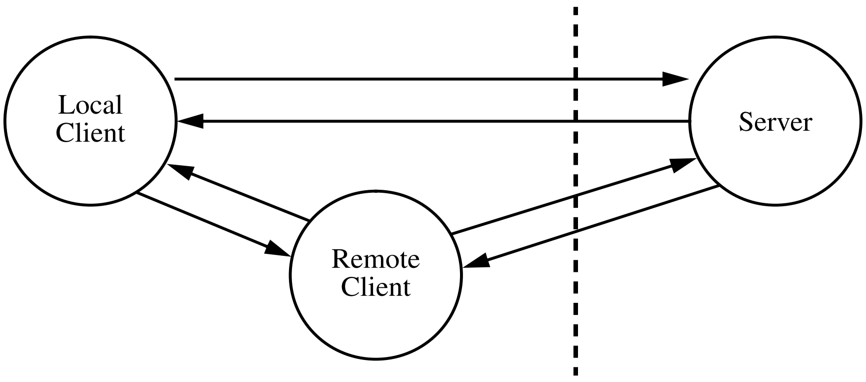


Figure 4.5: The 3-machine decomposition in multigrain shared memory systems. The dotted line indicates a node boundary.

on messages for communication between the different state machines. Together, these state machines and the messages they send and receive synthesize a uniform shared memory address space on top of a distributed memory architecture, support replication and caching of data, and maintain coherence on replicated data.

In conventional shared memory systems, there are two distinct types of shared memory state machines, a machine on the processor side and a machine on the memory side, as illustrated in Figure 4.4. The processor-side machine is responsible for handling events posted by the local processor and its primary cache, namely cache miss events, and for handling messages received from the memory-side machine for enforcing coherence, namely invalidation messages. The memory-side machine is responsible for distributing copies of shared data in response to data request messages from the processor-side machine, and for initiating coherence on replicated data when needed. One characteristic of this conventional 2-machine construction is that all shared memory transactions require the participation of both processor- and memory-side machines, and each transaction results in inter-node communication, as indicated by the node boundary dashed line in Figure 4.4. The non-local nature of all transactions in a 2-machine architecture was discussed in Section 4.1.2 and indicated in Table 4.1.

Contrary to conventional DSMs, multigrain shared memory systems are constructed using three distinct state machines. These machines are called the Local-Client machine, the Remote-Client machine, and the Server machine, as shown in Figure 4.5. Together, the Local-Client and Remote-Client machines are responsible for both maintaining coherence on mapping state and implementing the processor-side portion of the page-level DSM protocol. The Server machine implements the memory-side portion of the page-level DSM protocol (same as the memory-side machine in conventional DSMs).

A 3-machine construction arises in multigrain systems because of the three-way physical distribution of the following shared memory resources: the directory and home copy of a page on the memory-side SSMP, the cache copy of a page on the processor-side SSMP, and the page mapping cached in individual processor TLBs (physical distribution of the last two resources can only occur if the DSM node is a multiprocessor; hence, conventional DSMs built using uniprocessor nodes only require two state machines). The number of copies of each resource dictates the number of images of each type of state machine. For instance, a page has only one home copy; therefore, there is only one Server machine per page. Each SSMP can have a cache copy of a page, so there is a Remote-Client machine for each SSMP per page. And since any processor in an SSMP can map a page, each page has as many Local-Client machines as there are processors in the entire DSSMP.

All three state machines shown in Figure 4.5 communicate with one another, but only communication with the Server machine requires inter-SSMP messages since the Local-Client and Remote-Client machines are colocated on the same SSMP. Communication between Local-Client and Remote-Client can use efficient intra-node messaging interfaces provided within SSMPs, or hardware cache-coherent shared memory.

For expository purposes, we identify four basic transaction types supported by the three state machines illustrated in Figure 4.5. Each transaction type represents one or more possible shared memory transactions, but all transactions of the same type exercise the state machines in the same manner. Below, we describe the characteristics of these basic transaction types under unloaded conditions. The interaction between simultaneous transactions is more complex, and is the topic of Section 4.2.2.

The four basic transaction types are TLB Fault, Page Fault, Page Upgrade, and Release. Figures 4.6, 4.7, 4.8, and 4.9 show how each transaction type exercises the three state machines, respectively. The figures are organized in three columns, one for each state machine. Each column specifies actions performed by the corresponding state machine. An arrow between two actions indicates that the source action sends a message to a destination machine, invoking an action on the destination. Arrows that fan out (in Figure 4.9 only) indicate that an action sends one or more outgoing messages; arrows that fan in indicate that an action receives one or more incoming messages. Finally, annotations in italics refer to the state transition diagrams and tables provided in Appendix A. The italicized numbers (below actions) refer to state transition arcs, and the italicized text (above arrows) refer to MGS message names. A complete list of MGS message names along with a brief description of each message appears in Table A.4 of Appendix A. We describe each transaction type in detail below.

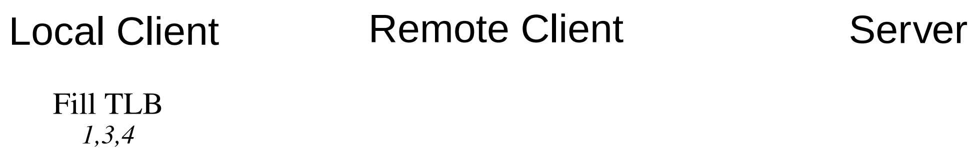


Figure 4.6: TLB fault transactions in MGS. Each column represents a state machine. All italicized text, *e.g.* the state transition numbers, refer to the state diagrams and state transition tables in Appendix A.

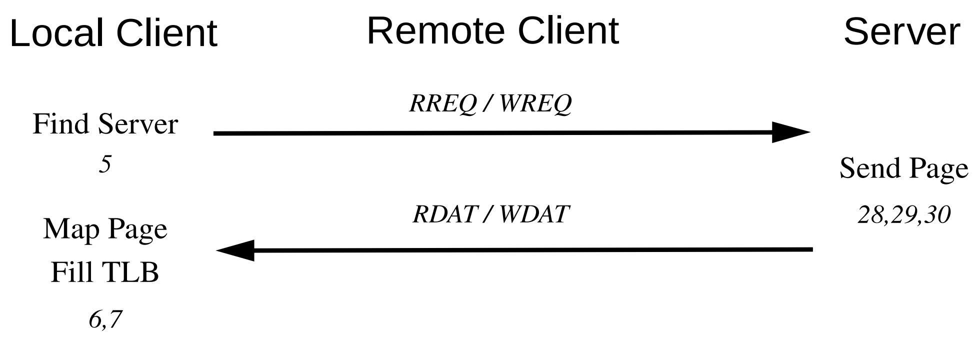


Figure 4.7: Page fault transactions in MGS. Each column represents a state machine. All italicized text, *e.g.* the message names and state transition numbers, refer to the state diagrams and state transition tables in Appendix A.

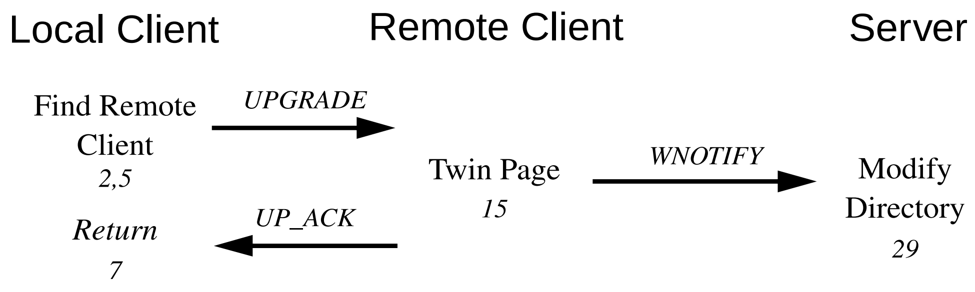


Figure 4.8: Page upgrade transactions in MGS. Each column represents a state machine. All italicized text, *e.g.* the message names and state transition numbers, refer to the state diagrams and state transition tables in Appendix A.

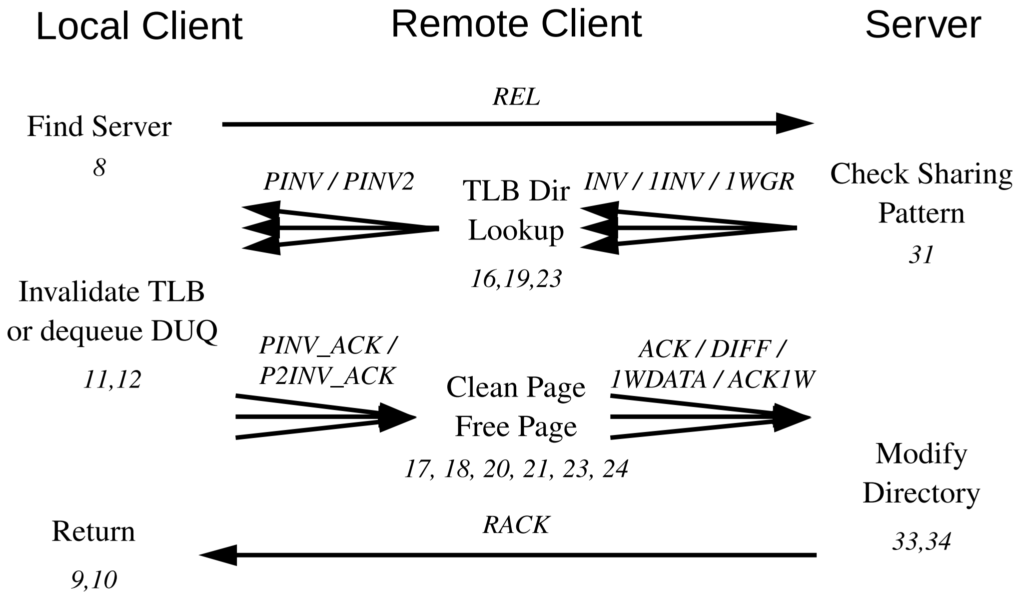


Figure 4.9: Release transactions in MGS. Each column represents a state machine. All italicized text, *e.g.* the message names and state transition numbers, refer to the state diagrams and state transition tables in Appendix A.

TLB Fault Transactions

The first type of transactions services TLB faults. These are the simplest of the four transaction types. All state accessed by the actions performed in these transactions are available to the Local-Client machine; therefore, assistance from the Remote-Client and Server machines is not necessary. The Local-Client simply accesses the page table in the local SSMP (see Section 4.2.3) and fills the TLB with the desired mapping entry.

Page Fault Transactions

Page Fault transactions require interactions between two state machines, the Local Client and the Server. These transactions service page faults, or accesses to pages for which the local SSMP has no copy. The Local-Client machine is responsible for locating the Server machine for the desired page, and sending a request for data to the page's Server. The Server machine is responsible for sending a copy of the page back to the Local Client. Page faults incur a round trip inter-SSMP messaging cost between the Local Client and Server, during which the Local Client is blocked.

Page Upgrade Transactions

Page Upgrade transactions arise when a processor attempts to write into a page for which the local SSMP only has read privileges. The Local Client initiates the transaction by sending a message to the Remote-Client machine responsible for the page on the local SSMP. The Remote Client performs a twinning operation on the read copy to make it writable. When this operation is complete, it acknowledges the Local Client (which is stalled during this time since it can't write the page until a twin has been made), and it sends a *WNOTIFY* message to the page's Server to notify it that an upgrade has occurred. Notice that the notification to the Server is unacknowledged. This provides a performance benefit because the Remote Client does not have to wait for a round trip to the Server; however, this benefit comes at some expense because it complicates the handling of simultaneous transactions. We will discuss this tradeoff in greater detail in 4.2.2.

Release transactions

Finally, Release transactions service requests for coherence initiated by the application. These transactions are the most complex, by far. The Local Client begins the transaction by sending a message to the page's Server. The Server consults the page directory and checks the sharing on the page. If the page has only one outstanding copy, indicating that sharing has been contained inside an SSMP, then the Server initiates the Single-Writer mechanism described in Section 4.1.2 by sending a *1WGR* (single-writer grant) message to the Remote Client. Otherwise, the Server initiates invalidation by sending an invalidation message to each Remote Client that has a copy of the page.

Each Remote-Client machine that receives a message from the Server consults a TLB directory and sends a message to each Local Client machine which has mapped the page. The type of message sent by the Remote Client depends on whether the Single-Writer mechanism or whether invalidation should be carried out. Under the Single-Writer mechanism, the Remote Client sends *PINV2* messages which cause the recipient Local Clients to perform a dequeue operation on their DUQs. The dequeue operation removes the DUQ element associated with the page, thus preventing any subsequent release operations from occurring. Under invalidation, the Remote Client sends *PINV* messages which cause the recipient Local Clients to invalidate the mapping for the current page from their TLBs. When each Local Client has completed its action, it acknowledges the Remote Client.

After the Remote Client receives acknowledgments from all the Local Clients, it in turn acknowledges the Server. For Release transactions that perform invalidation, each Remote Client cleans its copy of the page, sends an acknowledgment to the Server, and then frees the page from physical memory. Remote Clients with read copies simply send *ACK* messages. Remote Clients with write copies piggy-back updates onto their acknowledgments. The update either contains a diff (*DIFF* message), or the entire page (*1WDATA* message) when diff bypassing is used. For Release transactions that invoke the Single-Writer mechanism, the Remote Client does nothing to its copy of the page, and simply sends an *ACK1W* message to the Server. After the Server has received all acknowledgments from the Remote Clients, it merges all updates, if any, into its home copy of the page. When the merge is complete, it sends a *RACK* message to the Local Client that initiated the release, thus completing the transaction.

In Release transactions, the Local Client performing the release is blocked for the

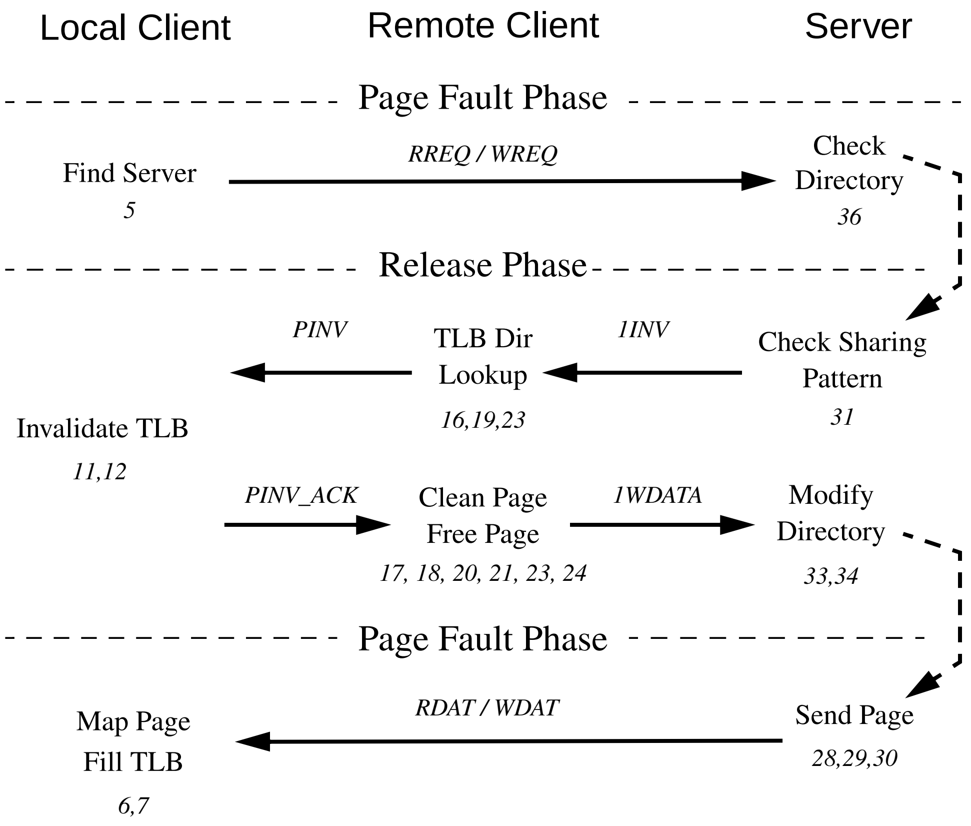


Figure 4.10: Single-Writer reversion as the compound of the Page Fault and Release transactions.

entire transaction. On the other hand, the Server exits after it sends messages to all Remote Clients; it is reinvoked only when acknowledgment messages arrive, once for each message. This split-phase approach minimizes the occupancy overhead associated with the Server machine. While we could also implement the Remote Client using split-phase transactions, in our design, the Remote Client waits for acknowledgments. We choose waiting because the small amount of work performed by Local Clients and the efficient communication mechanisms within SSMPs do not justify the overhead of exiting and re-invoking the Remote-Client machine for each acknowledgment message.

Other transactions

Aside from the four basic transaction types described above, there is one other transaction in MGS that handles reverting a page in the Single-Writer state back to the normal mode of coherence. Single-Writer reversion occurs when a page fault happens on a page that has transitioned to the Single-Writer state on a remote SSMP (see Section 4.1.2 for details of the Single-Writer mechanism). This transaction can be viewed as the compound of two existing transactions, the Page Fault transaction and the Release transaction; therefore, we do not consider this transaction as a separate transaction type.

Figure 4.10 illustrates how Single-Writer reversion can be constructed using a combination of the Page Fault and Release transactions. The page fault is intercepted by the Local Client and a message is sent to the Server, just as in a normal Page Fault transaction. the Server detects that the page is in the Single-Writer state. At that point, the Server defers completion of the Page Fault transaction and initiates the invalidation of the page, just like it would in a Release transaction. The invalidation proceeds normally. When all acknowledgments have been received (in this case, there is only a single *1WDATA* message), instead of completing the Release transaction, the Server picks up the Page Fault transaction exactly where it left off and returns a copy of the home, which is now coherent, to the faulting Local Client.

4.2.2 Simultaneous Transactions

Section 4.2.1 described in detail the operations performed during each shared memory transaction in the MGS system. This provides an understanding of how shared memory functionality is supported; however, it does not address the issue of correctness that arises when multiple transactions occur simultaneously. There are four issues related to simultaneous transactions that we will address in this section: synchronization within SSMPs, transient off-line states, single-writer condition violation, and unacknowledged upgrades.

Synchronization within SSMPs

Many simultaneous transactions involve conflicts within the same client. Specific examples include simultaneous TLB faults on multiple processors in the same SSMP, or a TLB fault simultaneously happening alongside an invalidation of the page data. Such simultaneous transactions are particularly frequent because of the number of competitors inside a client: the Remote-Client machine and one Local-Client machine per processor.

In MGS, we synchronize simultaneous transactions within clients by using shared memory locks. Each page has a single lock, called a *page lock*, in every client. Whenever a client-side action occurs, specifically TLB fault, page fault, page upgrade, page invalidation, and transition into Single-Writer mode, the machine performing the action (either the Local Client or Remote Client) must acquire the requisite page lock. Details concerning exactly when locks are acquired and for how long they are held are provided by the state transition table that appears in Appendix A

Transient Off-Line States

During a Release transaction that results in the invalidation of a page and all mapping state associated with that page, the data for the page is unavailable. This period begins when the Server machine receives a *REL* message, and does not end until all acknowledgments have been received from Remote Clients and all updates have been merged into the Server's home copy of the page. We call this transient period the "Release in Progress" state, labeled *REL_IN_PROG* in the state transition diagram of Appendix A.

When the Server machine enters the *REL_IN_PROG* state, it goes off-line with respect to all other requests for the page. This includes requests for data from page faults that happen simultaneously with the Release transaction, as well as other attempts to initiate coherence on the page.

In our design of MGS, transactions that occur while a Release transaction is in progress are deferred until the Release transaction completes[10]. Once the Server machine receives a *REL* message, it creates a *Deferred Transaction Queue* structure (see Section 4.2.3) and transitions into the *REL_IN_PROG* state. These actions happen atomically with respect to transactions coming into the Server machine. Any incoming transaction that finds the page in *REL_IN_PROG* state enqueues itself onto the *Deferred Transaction Queue* structure, providing enough information to allow the transaction to be completed at a later time. When the Server machine completes the Release transaction, it processes all transactions queued in the *Deferred Transaction Queue*.

Single-Writer Condition Violation

The previous section looks at problems with transactions that happen simultaneously with Release transactions resulting in invalidation. A similar problem arises when the release invokes the Single-Writer mechanism. When the Single-Writer mechanism is invoked, the Server machine transitions into the *REL_IN_PROG* state, and the page goes off-line for the duration of the transaction, just as it would if invalidation had been initiated; therefore, if simultaneously a request for data is received by the Server, the Server cannot process the request until the page leaves the transient *REL_IN_PROG* state. However, the Single-Writer case is more insidious than the invalidation case because the request for data violates the condition which initiated the Single-Writer mechanism in the first place (*i.e.* that there is only one sharer on the page).

The violation of the Single-Writer condition means that the system must revert the page out of the Single-Writer state and back to the normal mode of coherence. But before this can be done, the transaction that invokes the Single-Writer mechanism must be allowed to complete. And all of this must happen before the data request can be processed.

In MGS, we handle this simultaneous transaction in a fashion similar to what was explained for transient off-line states above. A *Deferred Transaction Queue* structure is created to defer the data request. When the Server receives the *ACK1W* message which completes the transition of the client's page into the Single-Writer state (see Figure 4.9), the Server will notice the deferred data request. Instead of acknowledging the Local Client which is blocked on a release operation (the release is what invokes the Single-Writer mechanism), the Server adds the release acknowledgment to the *Deferred Transaction*

[10] An alternative to deferring transactions is to respond with a busy message. Clients that are busied would be responsible for retrying the transaction at a later time. This, in fact, is the approach used in Alewife when a hardware directory receives a request for a cache line that is in the processor invalidation. In software shared memory, the cost of messaging required to busy the client outweighs the cost of tracking transactions that are received during transient states.

Queue, and initiates an invalidation on the page. The invalidation happens just as it would normally (see Figure 4.10), except that when it completes, it responds to both the deferred data request (with an *RDATA* or *WDATA* message) and the deferred release acknowledgment (with a *RACK* message).

Unacknowledged Upgrades

As we discussed in Section 4.2.1 above, *WNOTIFY* messages used to inform the Server machine of an upgrade action by the Remote-Client machine are not acknowledged by the Server. The consequence of this design decision is that for all outstanding read copies of a page, the Server machine cannot be certain whether the client really has a read copy or whether that read copy has been upgraded to a write copy and the Server has just not yet been notified. The incomplete information on outstanding read copies at the Server due to unacknowledged page upgrades slightly complicates invalidation when it occurs simultaneously with a page upgrade because of the difference in invalidation of read pages versus write pages—invalidated read pages return *ACK* messages which do not carry data, and invalidated write pages return *DIFF* or *1WDATA* messages (depending on whether diff bypassing is used) which do carry data.

To handle simultaneous invalidation and page upgrade transactions correctly, the Server machine must be prepared to receive a variable mix of read page and write page acknowledgments after it initiates invalidation. Let $\mid read\_dir \mid$ equal the size of the read copy set and $\mid write\_dir \mid$ equal the size of the write copy set at the time the Server initiates invalidation for some Release transaction. The total number of acknowledgments the Server can expect is fixed at $\mid read\_dir \mid + \mid write\_dir \mid$; however, the number of read acknowledgments can vary between 0 and $\mid read\_dir \mid$, and the number of write acknowledgments can vary between $\mid write\_dir \mid$ and $\mid read\_dir \mid + \mid write\_dir \mid$. Due to this variable mix of acknowledgment types, the Server machine must be careful in the allocation of *Diff Buffers*, structures used to store the data portions of incoming *DIFF* messages (see Section 4.2.3). In particular, it must ensure at invalidation time that enough *Diff Buffer* resources are available to handle the maximum number of write acknowledgments.

The design of the Page Upgrade transaction represents a tradeoff of slightly higher protocol processing overhead and complexity with communication overhead. The alternative to our design is to force the Remote-Client machine to wait for an acknowledgment before allowing its copy of the page to become upgraded. This implies the Local-Client must be blocked for an additional time equal to at least a round trip between the Remote-Client and the Server machines, which is fairly expensive since this communication occurs between SSMPs. We believe our design, which removes the need to block the Local-Client machine for a round trip at the expense of increasing protocol processing overhead on the Server machine, is favorable given the trend of increasing processor performance and the difficulty of decreasing communication latency.

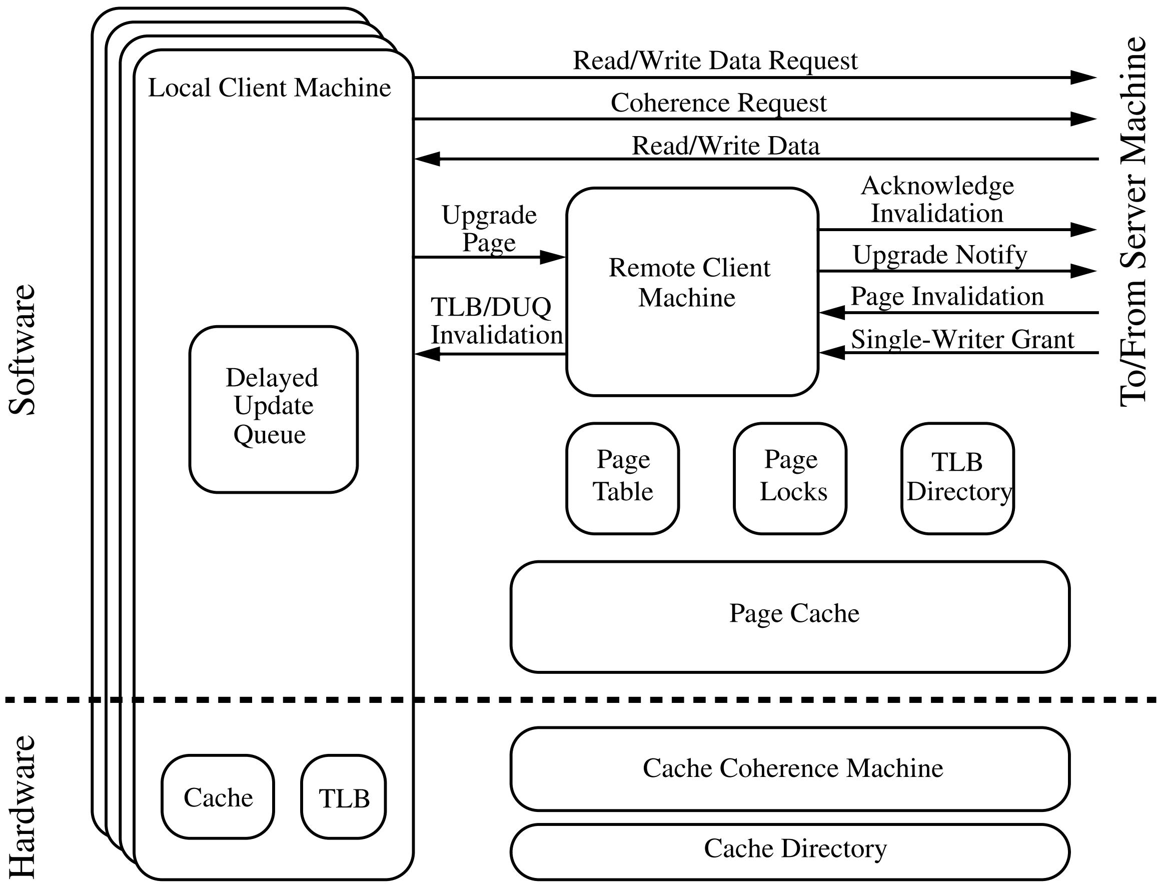


Figure 4.11: MGS Client architecture. Both hardware and software modules are shown. Arrows indicate different types of communication between the software modules.

4.2.3 Low-Level Components

In this section, we extend the discussion on the three state machines introduced in Section 4.2.1. While Section 4.2.1 described the interactions between the state machines, and how these interactions are composed to implement shared memory transactions, this section will focus on the data structures needed by the state machines to carry out their functionality.

Figures 4.11 and 4.12 show the components that form the Local-Client, Remote-Client, and Server machines, as well as those components that are accessed outside of the state machines. Arrows connecting the machines indicate different types of communication that occur between the machines, as described in Section 4.2.1. Where appropriate, we also indicate the division between hardware and software components using a dotted line.

We begin by describing the client, shown in Figure 4.11, which includes the Local-Client machines, the Remote-Client machine, and the components that are found on the client side of each SSMP.

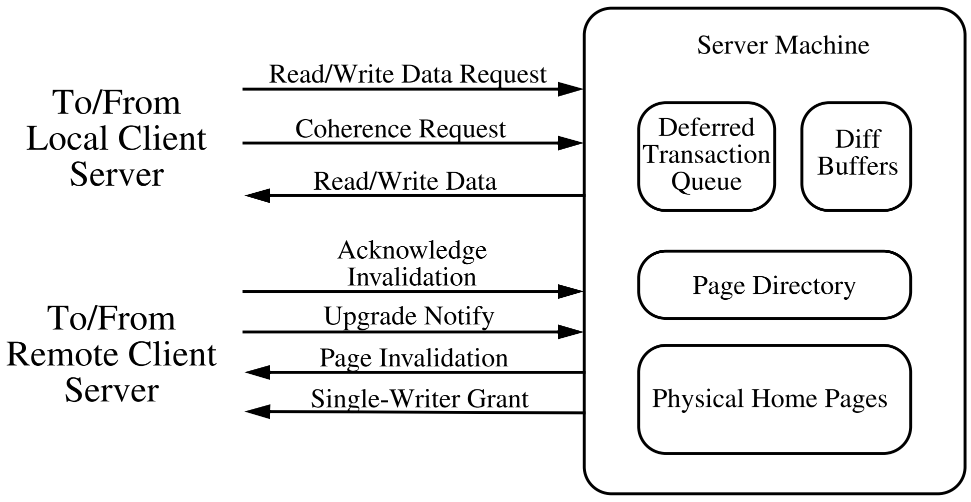


Figure 4.12: MGS Server architecture. All modules shown are software modules. Arrows indicate different types of communication between the software modules.

Hardware Components. There are four main hardware components, all of which reside in the Client. They are the hardware cache and TLB (one each per processor in the SSMP), and the Cache Coherence Machine and Cache Directory. These are the normal hardware structures one would find in a cache-coherent shared memory multiprocessor.

Local-Client Machine. There is a Local-Client machine for each processor in the SSMP. Whenever the Local Client is invoked, it executes on the processor to which it is assigned. This is clearly necessary for TLB invalidation since on most multiprocessor architectures, a processor's TLB is only accessible by the processor that owns the TLB. For page faults, handler code is executed by the faulting processor. This policy is reasonable since the faulting processor cannot make forward progress until the page fault has been processed; therefore, it may as well do the page fault processing. Furthermore, by making the faulting processor run the page fault handler, the placement of the faulted page will be in the faulting processor's memory module, *i.e.* a first touch page placement policy. Subsequent accesses made by other processors in the SSMP will occur remotely across hardware DSM nodes[11].

Remote-Client Machine. Unlike the Local-Client Machine, which is statically assigned one to each processor, the ownership of the Remote-Client machine, and thus the responsibility for processing, migrates from processor to processor within the SSMP. At any given time, the ownership of the Remote-Client machine belongs

[11]This is only an issue for SSMPs that assume a hardware DSM architecture, which is the case for our implementation of MGS. SSMP nodes that are symmetric multiprocessors all share the same memory modules, and therefore, there is no page placement issue.

to the processor that also owns the page for which processing is required[12]

Delayed Update Queue. There is one Delayed Update Queue (DUQ) structure for every Local-Client machine, and thus every processor in the SSMP. The DUQ is a list of pages that have been modified by the local processor. This list specifies exactly the set of pages that must be made coherent when the processor performs a release operation, as required by Release Consistency (in other words, on a release, the Local Client issues a coherence request for every page in the DUQ). The DUQ is the same structure that appears in the Munin system [13].

The Local Client adds an entry to its DUQ each time a page fault for a write access, or a TLB fault that results in an upgrade occurs. There are two instances in which entries are removed from the DUQ. First, if a page with read-write privilege is invalidated, the DUQ entry for the page is removed from all DUQs in the SSMP (along with address mappings for the page cached in the TLBs). Second, when a release operation on a page invokes the Single-Writer mechanism, a similar invalidation of DUQ entries occurs. Like TLB invalidation, DUQ invalidation is performed selectively by referring to the TLB Directory (see below).

Page Table. This software structure contains translation entries from virtual to physical frame numbers for all the pages that are resident on the SSMP. Each entry also contains the access privilege allowed on the page. Either read-only or read-write privileges are possible. This allows the software DSM protocol to treat read requests and write requests differently which is useful because the overhead for managing read-write pages is somewhat higher than for read-only pages.

Both the Local-Client and Remote-Client machines access the page table. The Local Client reads the table during TLB faults, and modifies the table during page faults. The Remote Client modifies the table during invalidation requests.

Many different page table organizations are possible [29]. The organization we choose for MGS is a bit unconventional because of particular constraints imposed by our hardware platform (see Section 5.2 for more details). However, in general, the page table used here is no different from any page table one would find in a normal operating system.

Page Locks. This is a pool of locks used by the Local and Remote Client machines to synchronize simultaneous transactions within an SSMP, as described in Section 4.2.2. Logically, there is a single lock for every possible entry in the page table. Because the total number of possible page table entries is large, the number of logical locks may become prohibitively high to implement physically. A simple

[12]Ownership of a page is clear in a hardware DSM–the owner of a page is the processor that owns the memory module in which the page resides. If the SSMP node is a symmetric multiprocessor, this definition does not apply since all processors physically share the same memory. In this case, we can define the owner of a page as the processor which performs the first touch on the page.

solution is to alias multiple logical locks onto the same physical lock. The degree of aliasing can be adjusted to trade off a smaller physical synchronization space for reduced concurrency.

TLB Directory. This software structure tracks page table entries that are cached in the TLBs of individual processors. It is an important structure in the implementation of TLB consistency. During page invalidation, the TLB directory is consulted to selectively interrupt only those processors that have the corresponding mapping cached in its TLB, thus avoiding unnecessary synchronization with processors that have not accessed the page.

Whenever the Local Client probes a page table entry into its TLB (during a TLB fault or after servicing a page fault), the corresponding TLB directory entry for the page involved is updated. Similarly, whenever the Remote Client performs a page invalidation, the corresponding TLB directory entry is cleared. Atomic access to this data structure is "piggy backed" onto the atomicity provided for page table access by the page locks. Each time the TLB directory is accessed, an access to the page table is made as well. By holding onto the page lock used for the page table entry while accessing the TLB directory, TLB directory accesses are guaranteed to be atomic.

Like the page table, this structure can become quite large. Yet, it is important that access to the structure remain efficient since all such accesses are performed inside a critical section. In MGS, the TLB Directory is implemented as a hash table.

The architecture of the Server is shown in Figure 4.12. The Server is implemented purely in software and therefore has no hardware components. Below, we describe each component in the Server in greater detail.

Server Machine. The Server is a state machine that executes the memory-side portion of the software DSM protocol. There is a single Server machine for each page in the system. Servers are statically assigned to SSMPs, and to an individual processor in the SSMP, based on the virtual address of the page associated with the Server. This static assignment is called the *home location* of the page, and is computed from page placement information supplied by the programmer through the `mgs_init_pagemap` interface (see Section 4.4 for details).

Physical Home Pages. The home location for a page is also the location where a permanent copy of the page resides. This copy, known as the *home copy*, represents the state of the page since the most recent coherence operation, or release, performed on the page (except when coherence is relaxed past release points as is allowed by the Single-Writer mechanism). The *home copy* is used to satisfy requests for data made to the Server machine by clients that wish to access the page.

Page Directory. This structure tracks replicated pages in the system. The Page Directory on a particular Server contains a single entry for every page whose home is on that Server. Each directory entry records all SSMPs that have a copy of the page, including whether the copy is in read-only or read-write state, and the PID of the processor that has current responsibility for the Remote Client machine (the latter is necessary so that the Server can invoke an invalidation request on the proper processor in the client). The directory also indicates whether the page is in normal coherence or Single-Writer mode.

Deferred Transaction Queue. As described in Section 4.2.2, there are transient off-line states during which the Server cannot respond to incoming transaction requests. Any requests received by the Server during these transient states must be deferred. The Deferred Transaction Queue is a data structure that tracks deferred transactions. For each incoming request received during a transient state, the Server records the client making the request, and the type of request (whether it is a request for read or write data, or whether it is a release operation). When the Server leaves the transient off-line state, it consults the Deferred Transaction Queue and processes all the recorded deferred transactions.

Diff Buffers. This is a pool of buffers that are used to hold diffs as they return from Remote Clients after an invalidation has been initiated but before the diffs have been processed. The Server machine processes diffs by merging the changes specified in each diff into the corresponding home page. Once all the changes in a diff have been merged into the home, the Diff Buffer can be returned to the pool for reallocation.

Because of the Diff-Bypass mechanism described in Section 4.1.2, the Server machine must wait for all outstanding diffs to return from the Remote Clients before diff merging can commence. The Diff Buffers are needed for storage of diffs waiting for processing by the Server. In our implementation of MGS, there is a fixed number of Diff Buffers that are available to each Server machine. The number of buffers is chosen to meet experimentally observed buffering requirements.

4.3 User-Level Multigrain Synchronization

MGS provides a user-level library that contains common synchronization primitives that can be called by application code. The library is separate from the main part of the system since it lives in the application layer instead of the communication layer. It is not a necessary component of the overall MGS system for multigrain shared memory functionality; however, it is an important complement to the MGS system since it delivers higher synchronization performance to applications that use the library. The synchronization primitives in the library achieve the highest possible throughput by leveraging information about the DSSMP architecture. Being cognizant of physical details in the system allow these primitives to significantly outperform a naive implementation.

While each synchronization primitive in the library works differently, the common theme in all the primitives is to limit the amount of communication during each syn-

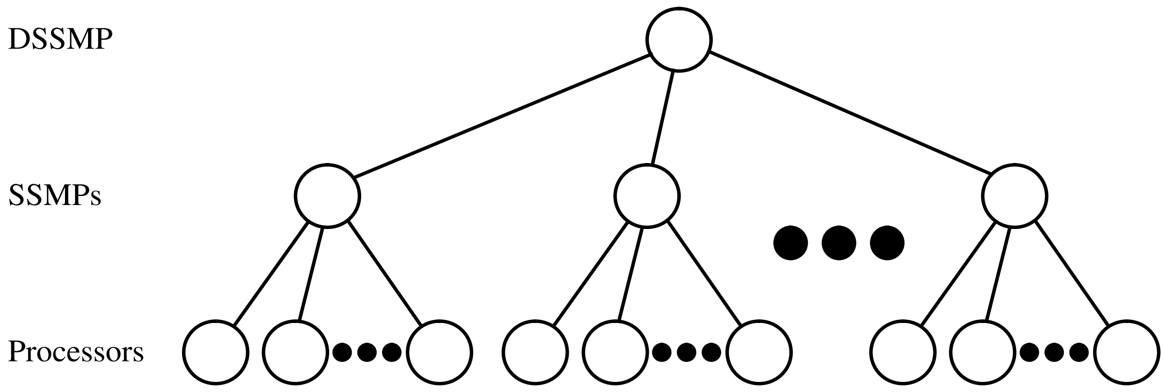


Figure 4.13: MGS Barrier.

chronization operation so that whenever possible, communication only occurs between processors within an SSMP, and communication between SSMPs is avoided. This is accomplished by creating a two-level hierarchy for each primitive such that there is a local primitive on each SSMP, and a single global primitive for the entire machine. If the primitive is designed properly, most synchronization operations will interact only with the local primitive, and only rarely will operations interact with the global primitive, thus minimizing inter-SSMP communication.

This general strategy has two benefits. First, it supports the communication requirements of synchronization operations using intra-SSMP communication mechanisms as much as possible rather than inter-SSMP communication mechanisms. This is beneficial because processors within an SSMP are more tightly coupled than processors between SSMPs and thus communicate more efficiently. Second, the hierarchical design of these synchronization primitives allow the system to leverage locality. If synchronization occurs repeatedly between processors within an SSMP, then communication across SSMPs can be avoided altogether. As we will see later in the thesis, locality of synchronization operations is a crucial attribute for applications to achieve high performance on DSSMPs because it directly minimizes software shared memory overhead (see Chapter 7). Building synchronization primitives that are more efficient in the presence of locality further improves performance.

The initial multigrain synchronization library built for this thesis includes two synchronization primitives, barriers and locks. We describe the details of these primitives in the following sections. Other primitives can be constructed using the hierarchical approach discussed above.

4.3.1 Barriers

The MGS Barrier is a message-passing tree barrier of depth three, as shown in Figure 4.13. Each level of the tree maps onto a portion of the DSSMP hierarchy: processor level (leaf nodes), SSMP level (intermediate nodes), and machine level (root node). The barrier works in the following manner. When a processor arrives at the barrier, it sends a message to the owner of the SSMP-level node in its local SSMP and waits. When the SSMP-level node receives a message from all processors in the SSMP, it sends a single message to the owner of the machine-level node and waits. When the machine-level node receives a message from all SSMPs, it initiates a release of the processors in reverse order (the machine-level node sends a message to each SSMP-level node, and in turn, each SSMP-level node sends a message to each processor-level node). Once a processor is released, it leaves the barrier and continues execution.

A tree barrier is efficient because it minimizes the number of messages sent between SSMPs. In the tree barrier, there are $2(\frac{P}{C} - 1)$ inter-SSMP messages, where P is the total number of processors in the DSSMP, and C is the size of an SSMP node. The multiplier 2 accounts for both the arrival message and the release message, and there are $\frac{P}{C} - 1$ messages on arrival and release because the root node is local to one of the SSMPs (the one to which it's assigned). Alternatively, a flat barrier would result in $2(P - C)$ messages (in this case, the root node is local to C processors, the number of processors inside an SSMP).

4.3.2 Locks

The MGS Lock is a hierarchical two-level lock consisting of a local shared memory lock, one for each SSMP, and a single global token-based lock implemented using message passing. Acquisition of the lock requires both acquisition of the local lock and ownership of a single token that is passed amongst the local locks. Figure 4.14 shows the design of the MGS Lock.

A processor first tries to acquire its local lock by performing an acquire operation on the mutex variable in its *Client Lock* structure. This acquire operation only competes with other processors from the same SSMP since the Client Lock is a per-SSMP structure. When the processor has successfully acquired the local mutex, it checks to see whether the token is "present" in the Client Lock. If so, the lock acquire operation completes. If not, the processor must steal the token away from the current owner of the token.

A steal request is initiated by sending a message to the owner of the *Global Lock* structure, indicated by the *home* field in the Client Lock structure. At the Global Lock, the mutex variable is acquired and the ID of the processor initiating the steal is inserted into the *Global Lock queue*[13]. Before the Global Lock mutex is released, the *Global Lock status* is checked. If the status is "BUSY," then a steal operation is already in progress and the Global Lock mutex is simply released. If the status is "FREE," then the status is changed to "BUSY" and a steal operation is initiated (the Global Lock mutex is also released to allow other steal requests to enqueue).

A steal operation is initiated by sending a message to the current owner of the token, indicated by the *owner* field of the Global Lock structure. At the Client Lock that owns

[13]Notice that there can only be one steal request initiated per SSMP because only one processor on each SSMP can successfully acquire the Client Lock mutex. Therefore, the processor ID inserted into the Global Lock queue in fact represents the SSMP that the processor belongs to.

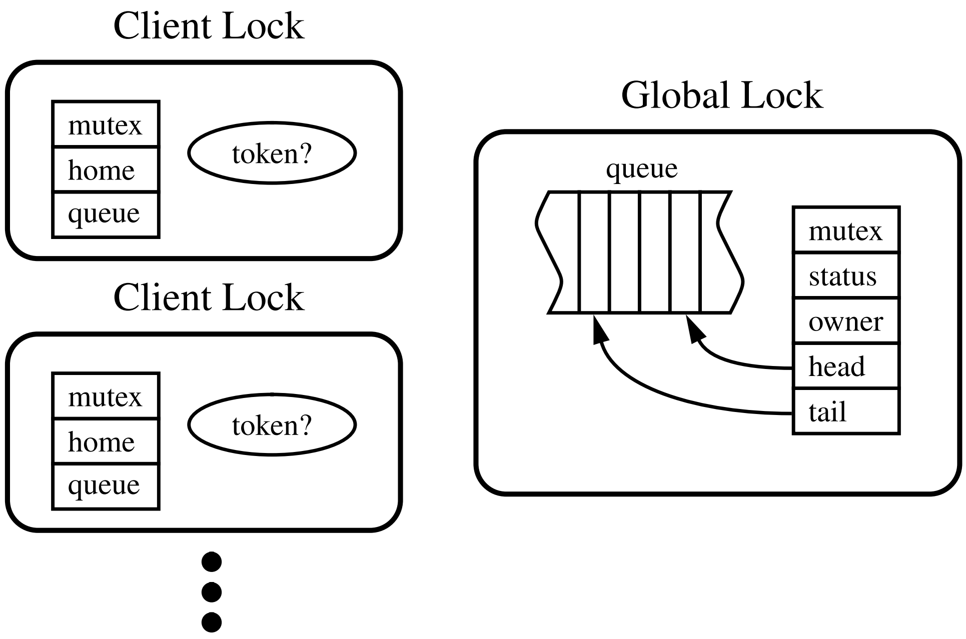


Figure 4.14: MGS Lock.

the token, the Client Lock mutex is acquired and the token is marked "not present." Then, a message is sent back to the Global Lock and the Global Lock mutex is re-acquired. The head of the Global Lock queue is dequeued. If the Global Lock queue is empty after the dequeue operation, the Global Lock status is changed to "FREE;" otherwise, the status remains "BUSY." Finally, the Global Lock mutex is released and a message is sent back to the processor to which the token is being passed, which is indicated by the value returned by the dequeue of the Global Lock queue. That processor marks the token "present" at its Client Lock, thus completing the lock acquire operation.

Notice there is one last case to handle. There must be a way to signal to the Client Lock when there are waiters still enqueued on the Global Lock queue immediately after a steal operation completes; otherwise, those waiters will be stranded forever. The mechanism for this is to mark the *queue* field in the Local Client after a token has been successfully stolen if there are other waiters left at the Global Lock. When a processor releases the MGS Lock, it checks the Client Lock queue field. If this field is marked, it forfeits its ownership of the token before releasing the Client Lock mutex thus waking up the processor that is enqueued at the head of the Global Lock queue.

The MGS Lock is very efficient if there is locality in the way synchronization operations are performed. Once a Client Lock owns a token, processors in the SSMP can repeatedly acquire and release the MGS Lock without communication with other SSMPs. Inter-SSMP communication occurs only if the token is subsequently stolen.

Function	Arguments	Returns
Memory		
mgs_init_pagemap()	int nprocs, int SSMP_size, int npages, int block_size	void
mgs_alloc()	int size	*char
mgs_release()	void	void
mgs_distribute_data()	void	void
Synchronization		
mgs_make_barrier()	**mgs_bar_t barrier_object	void
mgs_barrier()	*mgs_bar_t barrier_object	void
mgs_make_lock()	**mgs_lock_t lock_object	void
mgs_lock()	*mgs_lock_t lock_object	void
mgs_unlock()	*mgs_lock_t lock_object	void
Statistics		
mgs_stats_on()	void	void
mgs_stats_off()	void	void
mgs_clear_stats()	void	void
mgs_dump_stats()	void	void

Table 4.2: Programmer's interface to the MGS system.

4.4 MGS Library Interface

MGS exports a uniform shared memory programming model to the programmer that spans multiple multiprocessors. Therefore, the software layers that implement the MGS system are transparent to the programmer since programming for an MGS system is no different than programming for any other shared memory machine. There exists, however, a low-level interface to the MGS library routines that control many aspects of MGS functionality, such as DSSMP virtual clustering configuration, memory allocation, synchronization primitives, and statistics. Some of the library interface calls would not exist in a production system, such as the call for controlling virtual clustering. The rest of the calls are common system support calls that exist in most programming environments, and therefore can be hidden from the programmer by developing a set of macros that would target a specific programming environment. For completeness, we describe the MGS library interface in this section.

Table 4.2 completely specifies the MGS library interface routines. Each row in Table 4.2 lists a different routine, providing the routine's name, the parameters passed into the routine, and the value returned by the routine. In addition, the list of routines has been organized into three sections according to the three types of functionality provided: memory, synchronization, and statistics.

The memory routines allow the program to parameterize and interface with the shared memory layer. `mgs_init_pagemap` is used to configure parameters in the machine and shared memory layer, and is invoked at the beginning of program execution before any shared memory objects are created or accessed. The routine takes four parameters. The `nprocs` and `SSMP_size` parameters fix the DSSMP configuration by specifying the number of total processors and the number of processors per SSMP node, respectively. As we will discuss in Section 5.1 of Chapter 5, our implementation of MGS supports flexible machine configuration via a technique we call *virtual clustering*; therefore, it is necessary to provide a mechanism for specifying the `nprocs` and `SSMP_size` machine parameters. Normally, the system would bind these parameters at boot time; we choose to expose the parameters to the program and bind them at runtime. Our approach makes it easy to reconfigure the machine between program executions (*i.e.* we don't require rebooting the machine). The next argument, `npages`, specifies the total size of virtual memory in pages. Our implementation of MGS requires the programmer to inform the system of the maximum working set for the application. This is a restriction imposed by our implementation of virtual memory (see discussion in Section 5.3.1 in Chapter 5). The last argument to the `mgs_init_pagemap` routine, `blocksize`, specifies the interleaving pattern for mapping virtual pages onto physical nodes.

In addition to `mgs_init_pagemap`, there are three other memory routines—`mgs_alloc`, `mgs_release`, and `mgs_distribute_data`. `mgs_alloc` allocates `size` bytes of memory from the shared memory address space, and returns the address of the first byte. It is identical to the `malloc` call in the C programming language. `mgs_release` initiates a release operation in the shared memory layer. A processor issuing a release is blocked until all modifications made by the processor have been made visible to all other processors in the system. `mgs_distribute_data` distributes static global variables on the calling processor to all other processors in the system.

The synchronization routines access the primitives provided in the multigrain synchronization library, discussed in Section 4.3. `mgs_make_barrier` and `mgs_make_lock` create new barrier and lock objects, respectively. `mgs_barrier` executes a barrier, and `mgs_lock` and `mgs_unlock` are used to acquire and relinquish a lock, respectively.

Finally, the statistics routines control and access the statistics facility in MGS used to profile the MGS layer. `mgs_stats_on` and `mgs_stats_off` turn statistics on and off, respectively. These routines are used to exclude the statistics gathering on code for which profiling information is not desired (*e.g.* initialization code). `mgs_clear_stats` resets all statistics counters. Lastly, `mgs_dump_stats` prints the current value for all statistics to the standard output stream. Section 5.3.5 of Chapter 5 discusses the statistics facility in our MGS prototype in greater detail.

Chapter 5

Implementation

In this chapter, we present a prototype implementation of the MGS system introduced in Chapter 4. There are three major sections in this chapter. The first section argues for using hardware DSMs as effective platforms for studying DSSMPs. The second section describes the particular hardware platform used for our implementation, the Alewife multiprocessor. And the third section addresses the specific implementation issues involved when implementing MGS on Alewife.

5.1 A Platform for Studying DSSMPs

In Section 3.3.3 of Chapter 3, we presented a taxonomy for identifying DSSMP configurations that consists of two system configuration parameters: the overall system size, P, and the SSMP node size, C. For a given system size, P, varying the SSMP node size explores several different DSSMP configurations, all of which belong to the same family. What is important about such an exploration is that it looks at a spectrum of machines that trade off cost and performance by providing more or less hardware support for shared memory. Having the ability to study the entire spectrum is a powerful tool in understanding the behavior of DSSMPs[1]. However, being able to study the entire spectrum assumes that SSMP node size can be changed in a flexible manner. Accommodating such flexibility impacts the implementation. This section discusses the implementation issues for a prototype of the MGS system that arise due to flexible clustering.

One way to prototype the MGS system is to take a direct approach: procure a cluster of multiprocessors and build into their operating systems a communication layer that provides multiprocessor VM faults, TLB consistency, page cleaning, and software DSM with the appropriate multigrain mechanisms, as described in Chapter 4. We call this approach *physical clustering* because the cluster boundaries that partition the DSSMP are fixed in hardware. All the systems currently proposed in the literature that resemble MGS [22, 50] (and that we are aware of) use physical clustering.

[1] More justification will be given for this claim when we discuss our performance framework in Section 6.2 of Chapter 6.

The main advantage of physical clustering is that the resulting prototype resembles very closely a production system. There is no mismatch between the prototype and the target system being studied. The only mismatch that occurs is in the difference between the generation of hardware used to build the prototype and that which is postulated for the target, a problem encountered by any research endeavor that involves systems building. Measurements taken on a physically clustered prototype are fairly faithful to what one may expect to observe on the target system.

Physical clustering, however, makes it difficult to accommodate flexibility in varying system configuration parameters such as SSMP node size because the system configuration is fixed in hardware. There are two alternatives for varying SSMP node size in a physically clustered prototype. The first alternative is to have enough physical resources, both in the number of processors per SSMP and in the total number of SSMPs, to cover all possible configurations. Then, at configuration time (perhaps when the cluster is booted), only those SSMPs and those processors on each SSMP that are needed are "turned on." The second alternative is to physically reconfigure the cluster to match the configuration that is desired. This involves swapping processor modules between SSMPs each time a new configuration is created.

Significant problems exist in practice for both of these alternatives. The first alternative requires a tremendous amount of physical resources. As illustrated by Figure 3.5 in Chapter 3, the methodology for studying DSSMPs proposed in this thesis requires the SSMP node size to be varied from 1 to P, where P is the total number of processors in the DSSMP. At an SSMP node size of 1, there are P SSMPs, each with a single processor. At an SSMP node size of P, there is a single SSMP populated with P processors. To accommodate all of these configurations, we would need a multiprocessor cluster consisting of P SSMPs, each equipped with P processors. In other words, to study a DSSMP of size P would require P^2 physical resources[2]. Realistically, the quadratic scaling renders this approach prohibitive from a cost standpoint for even moderate values of P.

Unlike the first alternative, the second alternative does not require an exorbitant amount of physical resources. By swapping processor modules to re-populate SSMPs, a total of only P processors and P SSMPs are needed to study all configurations of a DSSMP of size P. However, there is the practical problem that each time the prototype is reconfigured, the system must be powered down and someone must physically rearrange the hardware. This can become onerous when many measurements are taken using many different configurations. It is also easy for the integrity of the hardware to be compromised when changes are made frequently.

The impracticalities of the solutions presented thus far all stem from the fact that the clustering configuration is implemented physically in hardware. In this thesis, we propose an implementation strategy that permits the highest degree of flexibility in varying SSMP

[2]This approach is viable if we constrain our methodology such that only certain DSSMP configurations are studied. For instance, if we constrain the maximum SSMP node size to some value smaller than P, then the required resources would be linear in P, where the constant of proportionality would be equal to the maximum SSMP node size allowed. This constraint, however, would limit the degree to which applications can be characterized.

node size, yet only requires a minimum amount of physical resources. The strategy we propose is called *virtual clustering*. In virtual clustering, the clustering configuration is not fixed in hardware. Instead, the desired clustering configuration is emulated by an all-hardware multiprocessor in which cluster boundaries are enforced in software.

Virtual clustering requires that the hardware platform is itself a distributed shared memory architecture. The distributed nature of hardware DSMs permit them to be partitioned in such a way that each partition has dedicated processor, communication, and memory resources. Other shared memory architectures, such as symmetric multiprocessors, do not have this property because communication (based on a bus) and memory resources are physically shared. The ability to partition hardware resources prevents different virtual SSMP nodes from competing for the same hardware resources. This is important if the prototype is to faithfully emulate the behavior of a DSSMP.

Clustering can be enforced on the hardware DSM by disallowing the use of hardware shared memory at virtual SSMP node boundaries. Such clustering can be achieved through intelligent management of the virtual memory system. Virtual memory provides a level of indirection into physical memory which, on a distributed shared memory machine, can be used to control which processors access which memory modules. Since a processor cannot access what it cannot name, isolation of shared memory traffic between virtual SSMP nodes can be achieved by allowing a processor to only map pages which reside in memory modules located within its virtual SSMP node. An attempt to access a page which does not exist within the virtual SSMP node should cause a page fault exception that is passed to the software DSM layer.

Our approach for emulating DSSMPs by building virtual clusters in software on top of hardware DSMs provides high flexibility in changing SSMP node size. Since clusters are defined in software, the clustering configuration can be changed trivially by setting a runtime parameter. Furthermore, our approach only requires the minimum amount of hardware resources. To emulate a P processor DSSMP, a P processor hardware DSM is needed. All possible configurations of the DSSMP can be studied by virtual clustering.

There are, however, some limitations associated with our approach. A prototype that emulates a DSSMP on a hardware DSM has some discrepancies as compared against a physical implementation of a DSSMP. For instance, support for communication between emulated SSMPs uses the same communication interfaces provided between hardware DSM nodes. Such communication interfaces will typically have higher bandwidth and much lower latency than the LAN networks used to connect SSMPs in a physical DSSMP. In Section 5.3.2, we will discuss a delayed message technique that makes the emulation of inter-SSMP communication more realistic. The delayed message technique artificially inserts delay into each inter-SSMP message using a hardware timer thus simulating a fixed delay for communication between SSMPs. Furthermore, the emulation approach requires a hardware DSM platform; therefore, each emulated SSMP will have a hardware DSM architecture. Again, there will be a discrepancy issue if the target system of interest is a DSSMP constructed from a cluster of bus-based symmetric multiprocessors.

5.2 The Alewife Multiprocessor

In this section, we discuss the hardware platform for our prototype of the MGS system, the Alewife multiprocessor [23]. The focus will be on aspects of the Alewife architecture, particularly those that impact the implementation of MGS. Details of how the implementation of MGS is carried out are deferred to Section 5.3.

Figure 5.1 shows a schematic diagram of the Alewife Machine. Alewife is a distributed memory multiprocessor that supports the shared memory abstraction and cache coherence in hardware. An Alewife core consists of a number of processing nodes connected in a 2-D mesh topology. The core consists of two parts: a compute partition and an I/O partition. The compute partition, denoted by circle-shaped nodes in Figure 5.1, is where user application code executes. The I/O partition, denoted by square-shaped nodes in Figure 5.1, provides access to external devices such as disks (not shown in the figure). The I/O partition occupies a small number of columns on one edge of the core mesh. Our prototype of the MGS system does not use the I/O partition, so we will not discuss it further. At one corner of the mesh, an interface to the VME standard I/O bus allows the Alewife core to communicate with a host workstation.

Each node in the Alewife compute partition consists of a SPARC integer core, called Sparcle [2], an off-the-shelf SPARC family floating point unit, 64K-bytes of static RAM that forms an off-chip first-level processor cache, 8M-bytes of dynamic RAM, the Elko series 2-D mesh router chip from Caltech (EMRC) [24], and the CMMU, Communications and Memory Management Unit, which synthesizes a shared memory address space across all the distributed memories, and implements a cache-coherence protocol. All chips on the Alewife nodes are clocked at 20 MHz[3].

As indicated in Figure 5.1, the 8M-bytes of dynamic memory are divided into three parts. The lowest portion of a node's memory is private to the node. This private area, which is 2M-bytes in size, contains the text segments for the operating system and user application. The middle portion of memory, also 2M-bytes in size, is managed by the CMMU hardware and stores the cache-coherence directories for all the shared memory locations home on the node. Finally, the last portion of memory forms the actual store for that portion of shared memory home on the node. This shared memory portion is 4M-bytes in size.

Two aspects of the Alewife architecture significantly impact the implementation of the MGS system: support for hardware cache-coherent shared memory, and support for fast inter-processor messaging. We discuss with these architectural features in greater detail below.

[3]The target clock speed of the Alewife machine is 33 MHz; however, due to a hardware bug in the first-step silicon, the machine is run at the slower 20 MHz speed.

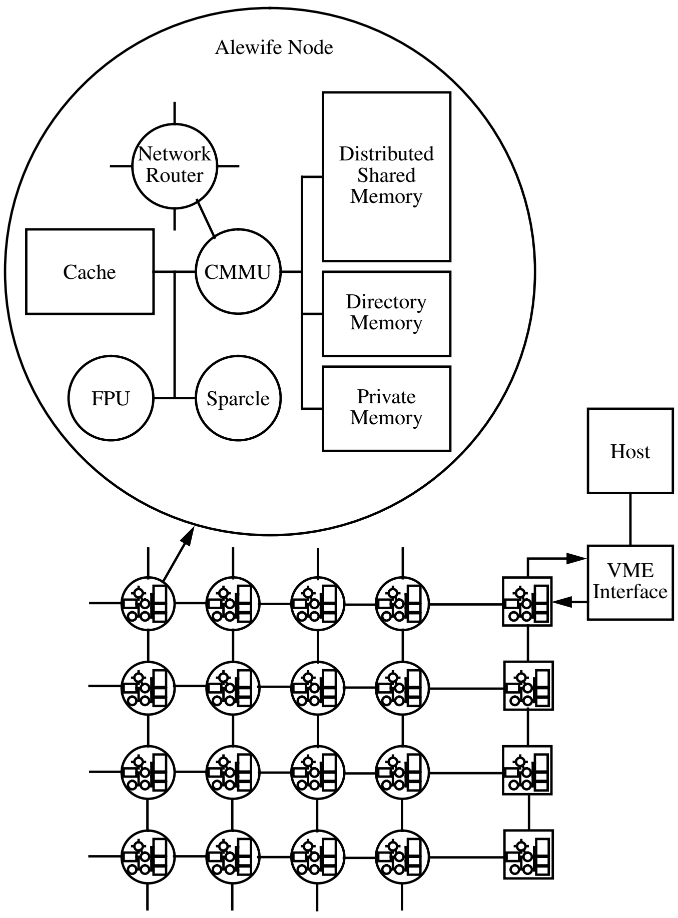


Figure 5.1: The Alewife Machine.

5.2.1 Hardware Cache-Coherent Shared Memory

The cache-coherence protocol in Alewife is a single-writer write-invalidate protocol that supports a sequentially consistent memory model [45]. The protocol uses a directory scheme [3] to track outstanding cache block copies in the system. This directory scheme, called LimitLESS [15], is based on a fixed hardware-managed directory structure that supports 5 pointers, but extends the hardware-managed directory to accommodate more pointers by trapping the home node processor to handle directory overflow in software. This software-extended approach is designed to handle the common case, small-degree sharing, efficiently in hardware, and to relegate the uncommon case, wide-degree sharing, to less efficient software.

LimitLESS directories impacts the MGS system by giving a performance advantage to DSSMP configurations with smaller SSMP nodes. In particular, configurations with virtual SSMP nodes of 5 processors or less are guaranteed to never pay the penalty of software directory extension since it is impossible in such an SSMP node for a cache line to be shared by more than 5 processors. However, if a virtual SSMP node contains more than 5 processors, then it is possible, particularly for those applications that exhibit wide-degree read sharing, for significant LimitLESS software overhead to negatively impact performance[4]. While intuitively larger SSMP nodes should lead to better performance, it is possible for DSSMPs with small SSMP nodes to outperform DSSMPs with large SSMP nodes because of the discontinuity in shared memory performance caused by LimitLESS.

In addition to lending a bias towards smaller SSMP nodes, LimitLESS can also reward a decrease in locality for applications running on a system with large SSMP nodes. Normally, if sharing on a memory object occurs solely between processors colocated in the same virtual SSMP node, higher performance is attained as compared to the same sharing pattern between processors on separate virtual SSMP nodes. This is because localized sharing patterns will benefit from the use of hardware shared memory, whereas sharing that crosses SSMP boundaries will incur software shared memory overheads. However, if the SSMP node size is large, then wide-degree sharing within an SSMP can invoke LimitLESS software. In this case, sharing patterns exhibiting less locality with respect to SSMP node boundaries may actually have a performance advantage. This is because when software shared memory replicates a page, in effect, the hardware directories are being replicated as well. Therefore, a wider degree of sharing can be handled after page-level replication leading to potentially less LimitLESS software overhead.

Finally, a comment should be made about the consistency model. As mentioned, Alewife supports sequential consistency. The software DSM layer, however, supports release consistency. The overall consistency model of the DSSMP is release consistency because RC is a weaker model than SC, and thus is the "limiting" consistency model. In general, as long as the consistency model supported by hardware shared memory inside each SSMP is as strong or stronger than release consistency, the memory model of the overall DSSMP remains RC.

5.2.2 Fast Inter-Processor Messages

Alewife provides architectural support for fast inter-processor messaging. Three hardware mechanisms, fast interrupts, multiple hardware contexts, and direct-memory access (DMA), and two software mechanisms, an active message model, and cached meta-process state, contribute to fast messaging support.

Sparcle, the Alewife integer unit, provides support for fast interrupts[5], a crucial mechanism for efficient message passing. In particular, Sparcle has a large interrupt vector space, and hardware support for dispatching different events to different vectors. The CMMU takes advantage of this large interrupt vector space. The message arrival interrupt on Alewife has a dedicated vector; it is one of the 16 asynchronous interrupt vectors that are supported by Sparcle. Whenever a message arrives and Sparcle is interrupted,

[4]In MGS, SSMP node size must be a power-of-two quantity. Therefore in practice, the breakpoint occurs while going from an SSMP node size of 4 to an SSMP node size of 8.

[5]In fact, the mechanisms we describe here are found on the SPARC processor architecture. Since Sparcle is derived from SPARC, it inherits the fast interrupt mechanism.

the processor begins execution of message handling code immediately. The latency of message handling is reduced since software dispatch code to figure out what interrupt has occurred is avoided.

In addition to fast interrupts, Sparcle provides another mechanism that helps reduce the latency of message handling, multiple hardware contexts. Multiple hardware contexts allow Sparcle to cache up to 4 threads of execution inside the processor. One use of multiple hardware contexts is fast context switching for latency tolerance [27]. In fast context switching, the processor switches between cached threads each time a thread suffers a cache miss to a remote memory module to hide the long latency of remote cache misses (the Sparcle processor can perform such a context switch in 14 cycles [2]). Another use of multiple hardware contexts is fast message handling. When a message arrives at a processor, it can process the handler associated with the message in a free hardware context, as long as one exists. This allows the processor to avoid save and restore overhead for the interrupted thread that would be necessary if only one context were available inside the processor.

Other mechanisms, supported in software, help reduce the latency of message handling on Alewife. Complimentary to multiple hardware contexts, the software messaging layer provides cached meta-process state for fast message invocation. For each hardware context in the Sparcle processor, a process block structure and a stack is allocated at boot time. When a message arrives at the processor, these cached software structures allow the message handler to execute immediately without suffering the overhead of allocation, as is necessary for general thread invocation. In addition, the message layer also supports the Active Message model [67]. Active Messages further reduce the latency of message invocation by providing the message handler address in the message itself. Because the message provides the handler address, the processor receiving the message can dispatch the message handler immediately without expending effort to figure out which message handler to execute.

Finally, the messaging interface on Alewife supports DMA transfers in hardware [42]. DMA allows the movement of bulk data through the messaging interface without burdening the processor with data movement overhead. This allows Alewife to support large messages very efficiently. DMA data in messages are *locally coherent*. Local coherence implies that data moved via DMA is coherent only with respect to the local memory module and local cache of the processor performing the DMA transfer. Coherence is not maintained with respect to the hardware caches of any other processor in the system. Such *global coherence* must be built on top of Alewife's DMA facility in software. This is the role of the page cleaning mechanism discussed in Section 4.1.2 of Chapter 4.

The Alewife mechanisms for message passing described in this section enable low overhead messaging, both in terms of low latency dispatching of messages when they arrive and transferring bulk data efficiently. These mechanisms have a significant impact on the MGS implementation. As Figures 4.11 and 4.12 from Chapter 4 indicate, significant communication occurs between the different MGS modules. All of these communications benefit from the support for low latency messaging[6]. Furthermore, those messages that carry bulk data, such as messages that provide read/write data and messages that respond to invalidation requests with updates, benefit from the support for DMA bulk transfer.

5.3 Implementation Issues on Alewife

In this section, we discuss several implementation issues that arise on Alewife. First, we address two problems facing the implementation of MGS: Alewife's lack of hardware support for address translation, and emulation of inter-SSMP messaging in a virtually clustered MGS prototype (see Section 5.1 for details on virtual clustering). Sections 5.3.1 and 5.3.2 deal with these issues, respectively. Then, we discuss how page cleaning, mapping consistency, and statistics gathering are implemented in MGS, in Sections 5.3.3, 5.3.4, and 5.3.5, respectively. Finally, Section 5.3.6 presents how our implementation of MGS is decomposed into user-level and kernel-level modules.

5.3.1 Software Virtual Memory

Alewife is a single-user single-program machine. Therefore, it does not provide traditional support for virtual memory (*i.e.* operating system support for page table structures and their management, and hardware TLBs for address translation and protection against unprivileged accesses). Since the software shared memory layer in MGS relies heavily on virtual memory support, MGS must build virtual memory on top of existing Alewife shared memory mechanisms in software.

There have been several schemes proposed in the literature for supporting virtual memory in software [57, 31, 5]. The general idea in software virtual memory (SVM) is that the compiler or the software system assumes responsibility for address translation and protection against unprivileged accesses in the absence of hardware TLBs. The compiler inlines translation and checking code before each access to mapped memory performed by an application. During a mapped access, the compiler-inserted inline code examines the virtual address of the access, reads the page table entry (PTE) associated with the address, and checks the access privilege specified by the PTE against the type of access being performed. If the check fails, the inline code signals an access fault to the operating system, and control is passed to the software shared memory layer which services the access fault. Otherwise, the check succeeds and the access is allowed to

[6]In fact, the support for messaging in Alewife is too good for those messages that cross virtual SSMP node boundaries (in Figures 4.11 and 4.12, those messages that provide communication between the MGS Client and the MGS Server). As discussed in Section 5.1, this mismatch can lead to optimistic performance results for the emulated DSSMP. In MGS, it is necessary to artificially slow down messages that cross virtual SSMP node boundaries to achieve higher emulation accuracy. This is the topic of Section 5.3.2.

perform. In this case, the inline code forms a physical address from the virtual address and its PTE, and the access is issued to the memory system.

While the general idea in SVM is simple, its implementation brings up some interesting issues. In this section, we will discuss three such issues: memory object management, efficiency, and the atomicity problem.

Memory Object Management

In an SVM system, the compiler must do two things. First, it must decide at compile-time which memory objects to place in virtual memory, and which memory objects to place in physical memory (for efficiency reasons described later, it is undesirable to place all memory objects in virtual memory). Second, for all memory references, the compiler must decide whether to inline translation and protection code before the memory reference code. The first responsibility is straight-forward. Any memory object which is guaranteed to be private to a single processor is managed in physical (unmapped) memory. All other objects, *i.e.* those objects that are shared by multiple processors and those objects for which sharing cannot be determined at compile-time, require coherence from multigrain shared memory and are thus managed in virtual (mapped) memory.

The second responsibility is somewhat more challenging for the compiler. Because there are two types of memory objects in the system, some memory references will use virtual addresses while others will use physical addresses. If the compiler can determine statically that the reference will always use a virtual address, then the compiler can inline translation and protection code before the reference. Similarly, if it is statically known that a reference will always use a physical address, the compiler can omit the inline code. However, a problem occurs if the compiler cannot determine statically what kind of address a reference will use. The primary example of this is pointers. For these references, it is not possible for the compiler to decide whether to inline or to omit inline code.

In MGS, we solve this problem by emitting the inline code for pointer references, but to include some checking code that determines at runtime whether a pointer contains a virtual address or a physical address. If the address is virtual, the checking code branches to the inline code, and translation and protection occurs on the address. If the address is physical, we branch around the inline code and issue the memory reference using the physical address immediately. This strategy assumes that virtual and physical addresses are easily distinguishable, a property that we will discuss further below.

Table 5.1 lists all the memory objects found in the MGS system. The column labeled "Management" indicates whether the compiler places the object in virtual or physical memory. The next column labeled "Static?" indicates whether static information allows the compiler to determine whether normal references to the object use virtual or physical addresses. And the last column labeled "Inlining" indicates the inlining strategy: "None" indicates no inlining code, "Trans" indicates inlining code for translation and protection, and "Check and Trans" indicates not only inlining code, but also checking code to see whether the reference uses virtual or physical addresses. In our implementation of MGS,

Object	Management	Static?	Inlining
Code	Physical	Yes	None
Static Variables	Physical	Yes	None
Stack Variables	Physical	Yes	None
Local Heap Variables	Physical	No	Check and Trans
Global Heap Variables	Virtual	No	Check and Trans
Distributed Arrays	Virtual	Yes	Trans

Table 5.1: Memory objects in MGS. The "Management" column indicates whether the object is managed in virtual or physical memory. The "Static?" column indicates whether the compiler can determine statically whether references to the object require inlining or not. The last column shows what inlining strategy is used.

only global heap and distributed arrays are shared across processors; these are the only objects in MGS that are placed in virtual memory. Heap variables, whether they are on the local heap or global heap, are referenced using pointers, and thus cannot be statically analyzed. While distributed arrays are referenced using pointers as well, these pointers are declared specially and are only allowed to point at distributed array objects. Therefore, the compiler has static information about them. Finally, code, static variables, and stack variables do not require inline code because they reside in physical memory, and the compiler can determine this statically. Local and global heap variables use pointers, thus checking code and inline code is necessary. Distributed arrays require inline code because they reside in virtual memory, but no checking code is necessary because the compiler can identify all references to these objects statically.

As mentioned earlier, one necessary property for our SVM strategy is that virtual and physical addresses are easily distinguishable. We achieve this through careful placement of the physical and virtual spaces used by MGS onto the Alewife physical address space. Figure 5.2 shows the memory map for the Alewife machine, and illustrates how MGS partitions the address space. Alewife has a 32-bit physical address space, where the high bit is used to differentiate between shared memory addresses and private memory addresses. All addresses with the high bit set are shared memory addresses; therefore, shared memory spans the addresses between `0x80000000` through `0xFFFFFFFF`. In our implementation of MGS, we partition this shared memory region such that the first 512 M-bytes (addresses `0x80000000` through `0xBFFFFFFF`) are allocated to physical (unmapped) memory, and the remaining 1.5 G-bytes (addresses `0xC0000000` through `0xFFFFFFFF`) are allocated to virtual (mapped) memory. The impact of this partitioning is that our implementation of MGS cannot run on an Alewife machine with more than 512 M-bytes of physical shared memory, otherwise the physical and virtual spaces will alias. 512 M-bytes of physical shared memory corresponds to an Alewife machine with 128 nodes (4 M-bytes of physical shared memory per node). We believe this is a reasonable limitation.

Placing the virtual space at the high end of shared memory makes it easy to identify virtual addresses: any address with the top two bits set is a virtual address. Figure 5.3 shows the pseudo-assembly code for detecting virtual addresses in the MGS system. Two

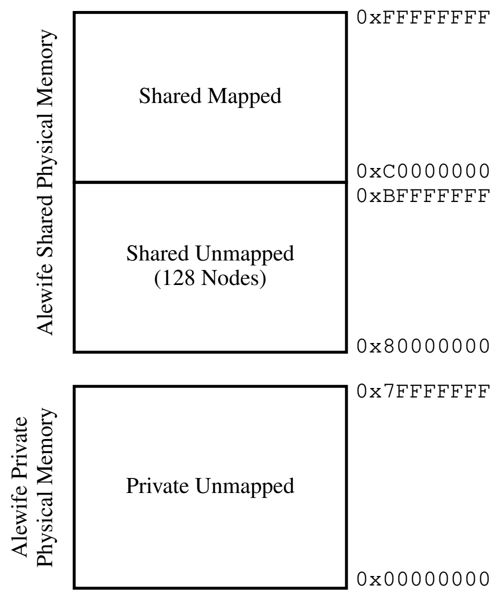


Figure 5.2: Memory map for the MGS system.

temporary registers, `rv1` and `rv2`, are used in the computation[7]. Register `rv0` is loaded with the address to be checked, `<addr>`, either from a register that already contains the address, or by calculating the address explicitly (i1). Register `rv1` is loaded with the constant `0xC0000000` (i2 and i3). The two registers are compared (i4), and if `rv0` is greater than or equal to `rv1`, then inline code is executed to perform a mapped access, otherwise an unmapped access is performed.

Efficiency

One of the main concerns in SVM is efficiency. SVM adds software overhead by inlining code before each mapped memory access which would otherwise be unnecessary if the system had a hardware TLB. The inlined code contributes to software overhead in two ways. First, extra processor cycles are expended in order to execute the inlined code. And second, inlining causes code expansion which negatively impacts cache performance and adds to register pressure. Because of the high frequency of memory accesses, these sources of overhead in SVM can become prohibitive if they are not addressed.

There are essentially two ways to reduce software overhead in SVM: reduce the frequency of accesses that require software inlining, or reduce the cost of the code itself inserted at each inline site. In our implementation of MGS, both of these overhead reduction techniques are applied. First, we reduce the frequency of inlining by using SVM only on memory references that have the *potential* for being shared, and therefore must be kept coherent through multigrain shared memory. As was shown in Table 5.1, many

[7]The Alewife Parallel C compiler [49] reserves two registers at every memory reference site for computation related to software virtual memory.

```
              i1: move <addr> --> rv0
              i2: move zero --> rv1
              i3: set-two-MSB rv1
              i4: compare rv0 rv1
              i5: br >= trans-label

              ** perform unmapped access **
              i6: jmp exit-label

trans-label:  ** perform mapped access **

exit-label:   i7: next instruction
```

Figure 5.3: Pseudo-assembly code for detecting virtual addresses in software virtual memory.

of the memory objects in MGS are placed in physical memory. If the compiler can determine statically that a memory reference will always reference an object managed in physical memory, then it can avoid inline code completely for that reference. Even if inlining is necessary, if the compiler can statically determine that the reference will always reference an object managed in virtual memory, then it can at least avoid the checking code (for instance, distributed arrays in Table 5.1).

Our implementation of SVM also tries to reduce the cost of the code at each inline site by using a simple page table structure. A significant portion of the cost associated with inline code for SVM is memory references to the page table. The page table structure is referenced to find the page table entry needed for address translation and access privilege checking. Depending on the type of page table structure used, several memory reads may be necessary before the PTE is found. For instance, in a forward-mapped page table structure with three levels of mapping, four memory loads are required before the PTE is obtained. If some of these memory accesses suffer cache misses, the cost of the inline code can be expensive.

To minimize the cost of accessing the page table, we implement a flat page table structure. In a flat structure, the page table is a one-dimensional array of PTEs, one PTE for every page in the virtual address space. Obtaining the desired PTE is simple because the structure is simple: index into the PTE array using the virtual page number. Therefore, with a flat page table structure, we can obtain the PTE with a single load instruction, since the location of the PTE is known once the virtual page number is known. Figure 5.4 shows the pseudo-assembly code for performing a mapped access (this is the code that would appear in place of the "** perform mapped access **" label in the checking code in Figure 5.3). As in the checking code in Figure 5.3, we again use two temporary registers, named `rv1` and `rv2`, to perform the computation. Also, we assume that at the entry into the mapped access code, the virtual address of the mapped access

```
trans-label: i1:  shift-right rv0 log(pagesize)        Compute virtual page number
             i2:  shift-left rv0 2

             i3:  load <page-table base>+rv0 --> rv0
             i4:  mask protection rv0 --> rv1
             i5:  compare rv1 <access-permission>       Check access permission
             i6:  move <addr> --> rv1
             i7:  trap-if-less-than <page-fault trap>

             i8:  mask page number rv0 --> rv0
             i9:  move <addr> --> rv1                   Calculate virtual address
             i10: mask page offset rv1 --> rv1          and perform access
             i11: vm-access rv0+rv1
```

Figure 5.4: Pseudo-assembly code for performing a mapped access.

resides in register `rv0` (which it does from the checking code in Figure 5.3).

As Figure 5.4 shows, we first prepare the virtual page number by right shifting the virtual address to eliminate the page offset field (i1). After the right shift, the virtual page number is obtained in `rv0`; we left shift this value by the size of the PTE (in our implementation, each PTE is 4 bytes, a single word in Alewife, so we shift by 2 bits) so that we can use the resulting value as an index into the flat page table structure (i2). The second block of pseudo code in Figure 5.4 performs the access privilege check. First, the PTE is loaded into `rv0` by using the virtual page number previously computed to index off of the base of the page table structure[8] (i3). The access privilege information is extracted (i4), and compared against the type of access being performed (i5). The result of this comparison is used to decide whether to signal an access fault condition (i7). The virtual address is placed in register `rv1` (i6) to tell the fault handler what address faulted in case the fault is signaled. The last block of pseudo code in Figure 5.4 performs the address translation and the actual access of data. First, the physical page number is extracted from the PTE (i8). Then, the virtual address is moved into register `rv1` (i9), and the page offset is extracted (i10). Finally, the access is issued whose effective address is the combination of the physical page number and the page offset (i11).

While using a simple flat page table structure allows for very efficient PTE lookups, the technique comes at a significant cost in space. A flat page table requires physical allocation of mapping state for the entire virtual address space, even if most of the address space is not mapped. This strategy does not allow for the dynamic allocation of page table state, such as allowed in a forward-mapped page table structure. To minimize the space cost of a flat page table structure, we only allow a portion of the addressable virtual space illustrated in Figure 5.2 to be mappable, and we allocate a flat page table structure only for that portion. This is accomplished by requiring the application to inform the

[8] The base address of the page table structure is always available in a special register on Sparcle.

MGS system the size of the virtual address space needed through the `mgs_init_pagemap` interface, as described in Section 4.4 of Chapter 4.

In Sections 6.1 and 6.3 of Chapter 6, we revisit the issue of efficiency in software virtual memory by quantifying the cost of software address translation.

Atomicity Problem

In conventional systems that support virtual memory, address translation and access privilege checking are performed by TLB hardware. In our implementation of MGS that supports virtual memory in software, address translation and access privilege checking are supported by expanding each mapped access into multiple instructions. One consequence of this code expansion is that the address translation, access privilege checking, and data access operations are not atomic as they are in conventional systems that support virtual memory in hardware.

The loss of atomicity due to inline code opens up the opportunity for mapping state (*i.e.* a page table entry) to become stale while it is being used by SVM code. This can be seen easily by looking at Figure 5.4 once again. Notice that there are several instructions that intervene between when a PTE is loaded (i3), and when the data access happens (i11) in the inline code that performs address translation and access privilege checking. We will call this sequence of instructions the *inline critical section*. If an invalidation of the PTE occurs while a processor is executing in the inline critical section, the PTE used by the inline code will become stale by the time the data access occurs. This can result in incorrect data being read, in the case the data access is a load, or a write to a bad location, in the case the data access is a store.

While the window of opportunity for an atomicity violation is very small (8 instructions as shown in Figure 5.4), the violation can nevertheless happen. The problem arises when a page invalidation request arrives exactly when a processor is simultaneously in its inline critical section for an address destined to the same page being invalidated. The handler that processes the invalidation request is invoked on the processor that is interrupted, and occupies the processor for the entire duration of the handler's execution. While the handler executes, the interrupted code does not make any forward progress and is not rescheduled to run until the handler completes. When the handler does complete, the PTE read at the beginning of the inline critical section will have been invalidated, and thus the inline code's copy will have become stale. When control is passed back to the inline code, the stale PTE will cause the incorrect data access behavior described above.

For correctness, it is imperative that the inline critical section be atomic with respect to page invalidation handlers[9]. However, any viable solution to this atomicity problem must pay close attention to the cost of the inline code. Because of its execution frequency, it would be unacceptable to add significant overhead to the inline code just for the sake

[9]Notice that atomicity is not required in the virtual address checking code shown in Figure 5.3 since this code only examines the virtual address of the current access and does not touch mapping state.

```
trans-label: i1:  shift-right rv0 log(pagesize)        Compute virtual page number
             i2:  shift-left rv0 2

             i3:  store <enter-critical> -->marker

             i4:  load <page-table base>+rv0 --> rv0
             i5:  mask protection rv0 --> rv1
             i6:  compare rv1 <access-permission>     Check access permission
             i7:  move <addr> --> rv1
             i8:  trap-if-less-than <page-fault trap>

             i9:  mask page number rv0 --> rv0
             i10: move <addr> --> rv1                  Calculate virtual address
             i11: mask page offset rv1 --> rv1         and perform access
             i12: vm-access rv0+rv1

             i13: store <exit-critical> --> marker

                                                       Inline Critical Section (i3–i13)
```

Figure 5.5: Pseudo-assembly code for performing a mapped access, with code to correct for the atomicity problem.

of providing atomicity. For instance, a solution that requires the inline code to acquire a lock before entering the inline critical section would be prohibitively expensive.

The solution we propose for the atomicity problem tries to be somewhat intelligent. We recognize that since the inline critical section is small, the frequency of invalidation requests landing inside this critical section must also be correspondingly small. Therefore, we will allow atomicity violations to occur instead of trying to prevent them from occurring. In the event that an atomicity violation does occur, we will place the burden of responsibility on the invalidation handler code to detect that an atomicity violation has occurred. Upon detection of an atomicity violation, the interrupt handler will roll-back the program counter (PC) of the interrupted background thread such that it points to the beginning of the inline critical section. Consequently, when the interrupt handler completes and the PTE has been invalidated, the interrupted thread restarts at the beginning of the inline critical section. It will then reread the PTE entry instead of using the stale copy that it read before the handler interrupt occurred. One requirement for this solution to work is that the inline critical section is restartable.

Our solution to the atomicity problem is attractive because it keeps most of the overhead out of the common case of inline code, and places it in the very infrequent case that a page invalidation interrupt occurs inside the window of opportunity for an atomicity violation. Of course, there is some impact on the inline code. Specifically, our solution requires that the invalidation handler can detect when an atomicity violation occurs. We require all inline code to set a marker in a predefined location in local memory whenever the processor enters an inline critical section. Then, it is possible for an invalidation handler to detect an atomicity violation–it simply checks the location to see whether the marker has been set[10]. Figure 5.5 shows the translation code example from Figure 5.4 with the marker code added. Two store instructions (i3 and i13) have been inserted to set and clear a marker location around the inline critical section. On Alewife, this adds a 6-cycle overhead (3 cycles per store instruction) to the inline code[11]. The figure also indicates those instructions that are inside the inline critical section.

5.3.2 Simulating Inter-SSMP Communication

As was discussed in Section 5.1, our implementation of MGS uses virtual clustering for flexibility in configuring the SSMP node size system parameter. The ability to change SSMP node size easily allows us to explore the entire spectrum of machines in any given DSSMP family, as discussed in Section 3.3.3 of Chapter 3. However, virtual clustering only emulates DSSMP behavior, and is less faithful to a target DSSMP system as compared against a physically clustered system. In this section, we discuss some of these discrepancies, and we describe techniques used in our implementation to make the emulation more accurate.

The problem with virtual clustering is the mismatch between the inter-SSMP communication interfaces used on the virtually clustered system and those used on an actual DSSMP. On a virtually clustered system, the communication interfaces used between SSMPs and within SSMPs are the same: they are the interfaces that are supported by the hardware DSM. These interfaces typically have much higher performance than those found on a DSSMP. Hardware DSMs typically employ special-purpose VLSI networks that tightly couple nodes. These networks are reliable and trusted; therefore, they do not require costly software protocol stacks to orchestrate end-to-end communication. For instance, in Alewife, after a processor constructs a message and injects it into the communication layer, the path the message takes from the sender's network interface, through the various routers in the network, and finally at the receiver's network interface is entirely in hardware. The lack of any system software in the communication layer allows messaging on hardware DSMs to be extremely efficient, both in terms of latency and bandwidth.

In contrast, the inter-SSMP communication layer in actual DSSMPs use commodity interfaces. The networks that connect SSMPs are commodity local area networks (LANs) such as ethernet or ATM. The communication interfaces at the sender and receiver are standard interfaces supported by the operating system, such as those from the IP family (*e.g.* UDP/IP or TCP/IP). The use of commodity interfaces significantly impact the communication performance between SSMPs that can be expected on a DSSMP.

---

[10]Notice that the invalidation handler only checks to see if it has interrupted an inline critical section. It is possible that an interrupted inline critical section is dealing with an address that has no relation to the page being invalidated. In this case, we still rollback the inline code to the beginning of the critical section. This is not detrimental to performance since rollback happens very infrequently.

[11]The 6-cycle overhead assumes the stores will hit in the cache. This is a good assumption since the same marker location is accessed each time inline code executes.

Delayed Messages

The first-order impact on performance due to the use of better communication interfaces on the virtually clustered system is simply that messages sent between SSMPs are delivered too fast. A very simple solution to address this problem, and the solution we adopt in our implementation, is to artificially delay the delivery of those messages that are sent between virtual SSMP nodes. This delayed messages approach simulates a fixed communication delay for all inter-SSMP messages.

Our implementation of MGS on Alewife provides a special message send primitive for inter-SSMP messages. It is the responsibility of MGS to decide which messages are sent across virtual SSMP node boundaries, and which messages remain within a virtual SSMP node. For inter-SSMP messages, the special message send primitive creates a bookkeeping data structure for the message that records everything needed to inject the message into the Alewife communication layer: the destination processor ID, the name of the message handler to be invoked on the destination processor, the number of arguments in the message, and the number of DMA regions to be sent with the message along with the DMA descriptors for each DMA region (starting address and length) necessary to perform the DMA. Once created, this bookkeeping structure is inserted into a pending message queue, and a timer is set for some fixed delay; the delay is specified as a runtime parameter to the MGS system. The timer counts down decrementing once every cycle starting from the fixed delay value, and when it reaches zero, it causes an interrupt. The timer interrupt dispatches to a routine that dequeues a message bookkeeping data structure from the pending message queue[12]. Using the bookkeeping data structure, the timer interrupt handler describes a message to the Alewife network interface and launches the message.

Two issues arise in the implementation of delayed messages. First, it is possible that when a timer interrupt occurs, it interrupts code that is in the process of sending a message itself. In this case, the Alewife network interface may contain a partially described message. The interrupt handler must unload the partial contents of the network interface into temporary storage, describe the message associated with the timer interrupt and launch it, and then restore the partially described message. However, the network output queue may be so full due to heavy messaging activity that there is not enough space in the queue to hold the message descriptors for both the message being sent by the timer interrupt and any partially described message left by the interrupted thread. In this case, the timer interrupt will postpone sending the message at the head of the pending message queue by resetting the timer[13] and exiting from the interrupt handler. The

[12]The bookkeeping data structure that is dequeued is the one at the head of the queue. The queue is managed in a FIFO order, so if there are multiple pending messages, the message at the head of the queue is the one that should be sent first. After a timer interrupt has been processed, if the pending message queue is not empty, the timer is reset for the new message at the head of the queue. The timer value used is the fixed delay value minus the time the new message has spent in the queue before reaching the head of the queue.

[13]The timer value we use to reset the timer is the same fixed delay value. It is possible to use a smaller

Message Type	Description	Consistency
1WDATA	Returns an entire dirty page from the client side to the server side. Responds to an invalidation request with the Diff-Bypass mechanism.	Reallocation of the page can cause data to be overwritten before data is sent.
DIFF	Returns a diff from the client to the server side. Responds to a normal invalidation request.	Reallocation of the diff buffer can cause data to be overwritten before data is sent.
RDAT WDAT	Sends page data from the memory side to the client side. Responds to a request for read or write data.	Data may change due to modifications by the home SSMP or by releases that happen before the data is sent.

Table 5.2: MGS messages that send data. The first column specifies the message type, the second column describes the function of the message, and the last column indicates the potential consistency problems associated with sending the data in a delayed fashion.

alternative would be for the timer interrupt handler to wait until the network output queue drains to provide enough queue resources. However, this is dangerous because timer interrupts on Alewife are served at a very high priority in the kernel; spin-waiting on the network at such a high priority can lead to deadlock. We have found in practice that the need to postpone messages due to a lack of network resources is extremely rare.

Another issue associated with the implementation of delayed messages is the consistency of data in those messages that carry application data (sent via DMA). By delaying the transmission of a message that contains application data, the data may change between when the message send is initiated and when the message is actually sent. We must ensure that such modifications to the data being sent will not cause the system to propagate errors. To examine the potential for data consistency problems, Table 5.2 lists all the messages in the MGS system that contain application data. The first column lists the messages (the message names correspond to those names used in Table A.4), the second column describes the function of the message, and the last column indicates the consistency problem that can occur if the message is sent in a delayed fashion.

The first two messages in Table 5.2, 1WDATA and DIFF, are both messages that send modifications from the MGS client to the MGS server in response to an invalidation request for a dirty page. After these messages are sent, the client deallocates the memory resource associated with the data. A problem can occur if this memory is reallocated before the message is actually sent. The processor to which the memory is reallocated can overwrite the contents of the data during the delay. We solve the consistency problem for 1WDATA and DIFF messages by delaying the deallocation of the memory resources associated with the data until after the message has been actually sent[14].

value, but we have not attempted to do so.

[14]Our special message send primitive provides a cleanup facility. The sender can specify a callout

The last two messages in Table 5.2, RDAT and WDAT, send page data from the MGS server to the MGS client in response to read and write requests, respectively. Home copies continually undergo modifications from two separate sources. Modifications can come from writes performed by processors in the same SSMP as the home copy. Also, modifications occur during release operations as updates from dirty pages are merged into the home copy after invalidation. Because of these modifications, the page data in an RDAT or WDAT message can be different from the contents of the page when the message send was originally initiated. Fortunately, this does not cause any data consistency problems due to the relaxed access ordering guarantees provided in the Release Consistency memory model. In RC, shared memory access ordering needs to be guaranteed only for special accesses [25] (acquires and releases). The ordering on acquires and releases are ensured explicitly by proper synchronization at the application level. The ordering of all other shared memory accesses can be arbitrary without violating the semantics of Release Consistency. Therefore, even if a modification occurs to a location inside a page that has been delayed for transmission in an RDAT or WDAT message, no consistency problem arises because RC allows the sending of the page data to happen in any order with respect to the modification.

Other Simulation Issues

Our delayed message approach partially addresses the mismatch in the inter-SSMP communications interfaces between a virtually clustered system and an actual DSSMP. If all inter-SSMP messages in an actual DSSMP implementation experience fixed delay, our approach would emulate the target system perfectly. Unfortunately, the fixed messaging delay assumption is only true for ideal systems. Actual DSSMPs exhibit much more complex behavior.

Variability in latency for the delivery of messages can come from many sources. First, message length effects the latency experienced by a message. Longer messages generally require more processing in various software protocol stacks associated with the standard communications interfaces provided between SSMPs in a DSSMP. Common protocol stack operations, such as data copying and checksum computation, increase linearly in cost as message length grows. In addition, longer messages have higher associated data transfer costs through network interface hardware and the physical layer of the network. Furthermore, most commodity communications interfaces place an upper limit on the maximum length of a network packet; therefore, transmission of a message through the network that exceeds this upper limit requires that the message be fragmented at the sender into smaller units that are each within the maximum packet length. A fragmented packet also requires reassembly of the fragments at the receiver to recover the original message. Both fragmentation and reassembly add protocol processing overhead along the critical path of message latency. In our implementation of MGS, we do not account

procedure to the timer interrupt handler that actually sends the message. The callout procedure is invoked by the timer interrupt immediately after the system sends a delayed message.

```
for (i = 0; i < page_size; i+= cache_line_size) {
      value = load(start_addr, i);
      store(value, start_addr, i);
      flush(start_addr, i);
}
```

Figure 5.6: Pseudo-C code for performing page cleaning. "start_addr" is the base address of the page being cleaned, "page_size" is the number of bytes in a page, and "cache_line_size" is the number of bytes in a cache line.

for variable message latency due to message length.

Perhaps an even more significant source of message latency than message length is contention. Given an application workload, the amount of contention that inter-SSMP messages experience will depend on the type of commodity communications interfaces used between SSMPs. In general, there can be several points along the path of an inter-SSMP message involving physical resources that can become points of contention. If the commodity communications interface requires software at (both sender and receiver) endpoints of communication for processing protocol stacks, then there may be contention at the protocol processors. Another place where contention can occur is at the hardware interface between the SSMP and the inter-SSMP network. Finally, contention can also occur at the routers within the inter-SSMP network itself.

Contention through the hardware network interface between processors on an SSMP and the inter-SSMP network can be a problem especially on SSMPs where the amount of bandwidth available through the network interface is fixed, regardless of the SSMP node size [22] (see Section 8.2 of Chapter 8). In such SSMP architectures, the fixed network interface bandwidth resource becomes a bottleneck as the SSMP node size is scaled since larger SSMP nodes generally place a greater demand on messaging into and out of the node. In our prototype implementation of MGS, the bandwidth between virtual SSMP nodes increases with SSMP node size. Due to the mesh topology of the Alewife multiprocessor, the number of network links available to a virtual SSMP node goes as the perimeter surrounding the node. This perimeter increases as the square root of the number of processors in the virtual SSMP node because of the two-dimensional nature of Alewife's mesh network. Being able to scale inter-SSMP bandwidth with SSMP node size requires scalable communications interfaces for SSMPs. In [34], the design of such a scalable communications interface, based on standard Internet protocols, is proposed and implemented with the MGS system. Experiments are conducted to investigate the effects of contention between inter-SSMP messages on application performance.

5.3.3 Page Cleaning

In Chapter 4, Section 4.1.2, we discussed the purpose of the page cleaning mechanism. In this section, we discuss how page cleaning is implemented as efficiently as possible on Alewife.

The purpose of page cleaning is to localize all data inside a page that has been

```
/* prologue loop */
for (i = 0; i < 4*cache_line_size; i += cache_line_size) {
    write_prefetch(start_addr, i);
}

/* main loop */
for (i = 0; i < page_size - 4*cache_line_size; i+= cache_line_size) {
    value = load(start_addr, i);
    store(value, start_addr, i);
    flush(start_addr, i);
    write_prefetch(start_addr, i + 4*cache_line_size);
}

/* epilogue loop */
for (i = page_size - 4*cache_line_size; i < page_size; i += cache_line_size) {
    value = load(start_addr, i);
    store(value, start_addr, i);
    flush(start_addr, i);
}
```

Figure 5.7: Pseudo-C code for performing page cleaning with prefetching optimizations. "start_addr" is the base address of the page being cleaned, "page_size" is the number of bytes in a page, and "cache_line_size" is the number of bytes in a cache line.

distributed to processors within an SSMP via hardware cache coherence. In our implementation of MGS, we perform page cleaning purely in software. A processor that wishes to clean a page explicitly walks down the page. For each cache line in the page, the processor performs a store operation to the cache line. Because Alewife supports a single-writer write-invalidate protocol, the store issues an invalidation for that cache line if there are outstanding copies in the SSMP. Once the store completes, we are guaranteed that there is an exclusive copy of the cache line in the system belonging to the processor performing the store. By flushing this copy after the store completes, we are guaranteed that there are no outstanding cached copies in the system. Figure 5.6 shows the pseudo-C code for this operation. An additional load at the beginning of each iteration of the loop is necessary because the store operation must store the same value that was originally in the page, otherwise data would be destroyed.

The approach shown in Figure 5.6 can suffer from large amounts of memory stall, thus decreasing performance. Most of the time in each loop iteration is spent waiting for the memory system to perform the necessary invalidations in order to provide an exclusive copy of the cache line to the processor performing the page cleaning. To address this performance bottleneck, we employ prefetching in order to hide the memory latency associated with invalidation. Figure 5.7 shows the same page cleaning code in Figure 5.6 augmented with prefetching.

Each iteration of the "main loop" in Figure 5.7 has the same load-store-flush sequence as before. In addition, a prefetch instruction is added at the end of the loop body to initiate the fetch of data that will be needed by the load-store-flush sequence several iterations ahead. In the example in Figure 5.7, the *prefetch distance* is 4 iterations. The prefetch instruction requests an exclusive copy (*i.e. write prefetch*), so the prefetch not only brings a copy into the requesting processor's cache, but it also forces any necessary invalidations as well. If the prefetch is successful, both the load and store operations that access the cache line 4 iterations later will find an exclusive copy of the cache line in the local processor's cache thereby completely masking the latency of the memory system. Figure 5.7 also contains a "prologue loop" and an "epilogue loop." The prologue starts up the prefetches before any stores are issued, and the epilogue performs the last load-store-flush sequences after the last prefetch is issued.

Notice that the prefetch does not eliminate the need for the load and store instructions in the loop body. These instructions are still necessary to guarantee that the prefetch completes. In most all shared memory systems (including Alewife), prefetches are only hints to improve performance; they can be ignored by the memory system without violating the cache-coherence protocol. Alewife drops prefetches under certain circumstances, such as when the prefetch request finds the cache directory busy (*i.e.* another transaction is in progress for the same cache line). The load and store instruction ensure that any necessary invalidations occur even if the memory system drops the prefetch request.

Other Page Cleaning Optimizations

The pseudo-C code in Figure 5.7 is the strategy used by our implementation of MGS for page cleaning. There are some other possible optimizations that we do not employ in our implementation that we will discuss briefly in this section.

As mentioned in Section 4.1.2 of Chapter 4, page cleaning provides a coherent copy of a page in the physical memory of the SSMP by localizing data that has been distributed via hardware cache coherence. This is necessary particularly for DMA devices that cannot move data coherently with respect to processor caches. Page cleaning, however, is unnecessary when there is no coherence issue, *i.e.* when there is no dirty cache line from a page outstanding in any processor cache. While this condition is difficult to detect in general, it is guaranteed if the page is only read mapped by the processors in the SSMP. Therefore, invalidations of read mapped pages can avoid page cleaning altogether.

While this optimization removes the overhead of page cleaning from the critical path for invalidation of read-only pages, it does not eliminate the need for page cleaning. In particular, such read-only pages cannot be reallocated before they are cleaned. As was pointed out in Section 4.1.2 of Chapter 4, reallocation of a page that has outstanding copies in processor caches can lead to the access of stale data. The benefit of this optimization, however, is that the page cleaning can be delayed and performed at a less critical time. For instance, read-only pages that have been invalidated by the MGS system can be placed on a pending queue. The operating system can then clean the page in the background when it finds there are spare cycles, and return the page to the free queue.

Pages that are mapped read-write cannot avoid page cleaning at the time of invalidation because it is possible for one or more processor caches to contain a dirty cache copy. However, one possible optimization for read-write pages is to clean the page selectively on a cache-line by cache-line basis. In our implementation of MGS, the processor performing the page cleaning is the home for the page being cleaned; therefore, it has access to the hardware cache directories for all cache lines in the page. During page cleaning, it is possible to first read the directory for each cache line being cleaned, determine whether the cache line is in read or write mode, and only clean those cache lines that are in write mode. Just as was the case for read-only pages, such partially cleaned pages cannot be reallocated to MGS until they are fully cleaned.

Partial page cleaning has the potential to save processor cycles, though evaluation on an actual implementation is necessary before drawing any conclusions. Not only is it necessary to clean fewer cache lines, but only those cache lines in write mode are cleaned. This is beneficial because cache lines in write mode, on average, can be cleaned with less latency than cache lines in read mode. A write mode cache line is guaranteed to have only one outstanding copy. Read mode cache lines have multiple-degree sharing, thus requiring more invalidation messages. And on Alewife, as stated in Section 5.2.1, when the degree of sharing exceeds 5, the cache line is managed under software which can increase the invalidation time by an order of magnitude. These savings provided by partial page cleaning must be balanced against the added overhead of consulting the directory for every cache line in the page.

5.3.4 Mapping Consistency

Figure 4.9 of Chapter 4 shows the actions performed by the MGS state machines on a release transaction. Part of the release transaction requires invalidation of mapping state for all the processors on a client SSMP. In Figure 4.9, this is indicated by the *PINV* messages that are sent from the Remote Client machine to the Local Client machines. The implication of these messages is that invalidation of mapping state is performed by the Local Client machine on each processor that has mapped the page being invalidated. In general, mapping state invalidation must be performed by the Local Client because it requires probing a processor's hardware TLB for the address mapping in question, an operation that can be performed only by the processor with the TLB.

In our prototype implementation of MGS, the invalidation of mapping state is performed by the Remote Client instead of the Local Client; therefore, our implementation never sends *PINV* messages, and handler code to perform mapping consistency is never invoked on the Local Client machines (though we do use the *PINV2* message to invoke handlers on Local Clients to perform DUQ invalidation–see Section 4.2.1 of Chapter 4). This is possible because our implementation of MGS supports virtual memory in software. In SVM, address mappings are cached in memory, not hardware TLBs. Furthermore, the mappings for each processor are placed in a portion of shared memory that is local to that processor on the Alewife machine. Consequently, address mappings are visible to all processors on each SSMP via cache-coherent shared memory. The Remote Client can invalidate an address mapping by simply performing a shared memory write to the location in shared memory where the address mapping resides.

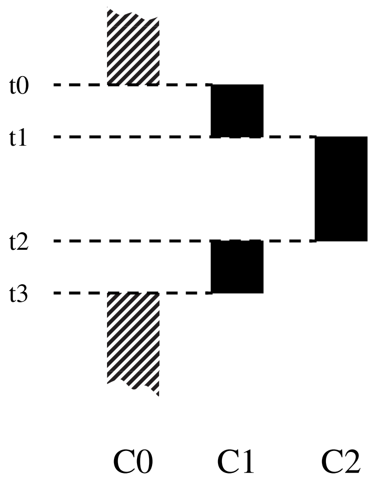
There are two implications of our implementation of mapping consistency. First, maintaining mapping consistency in our system is cheaper than in a system that supports virtual memory in hardware. While our system uses efficient hardware shared memory mechanisms to invalidate mapping entries, a system that caches address mappings in processor TLBs requires a software implementation of mapping consistency that uses inter-processor interrupts to invoke Local Client handlers. Second, the solution to the atomicity problem described in Section 5.3.1 cannot be used to enforce atomicity of the inline critical section against mapping invalidation on the Local Client machines. There is no way for the Remote Client to detect that another processor on the same SSMP is executing the inline critical section when address mappings are invalidated through shared memory. In practice, we have never seen an atomicity violation occur on the Local Clients during a page invalidation on the Remote Client. This is because the inline critical section performed on each Local Client is so short (8 instructions as shown in Figure 5.4) relative to the latency between the invalidation of mapping entries and the actual invalidation of the page performed on the Remote Client (on the order of 1000s of instructions). Our implementation, however, still uses the PC rollback solution to enforce translation atomicity on the Remote Client. Atomicity violations are frequent on the Remote Client because the page invalidation handler (which also performs address mapping invalidation) occupies the processor executing the Remote Client for the entire duration of the handler. Consequently, an atomicity violation is guaranteed to occur as long as the interrupt for the page invalidation handler occurs inside an inline critical section (see explanation in Section 5.3.1.)

5.3.5 Statistics

Many different types of statistics are collected on the MGS system to produce the experimental results that appear in Chapter 6. All of the statistics were gathered using the four hardware cycle timers provided on the Alewife machine. Statistics instrumentation code is inserted into the MGS system code to turn the Alewife timers on and off at the appropriate times, and to read the timers in order to acquire the cycle counts. While such software instrumentation is intrusive, the impact on end runtime is negligible. Most of the statistics gathered for Chapter 6 are trivial to implement; however, one statistic, MGS runtime overhead, requires system support. We describe this statistic in greater detail in the rest of this section.

The MGS runtime overhead statistic counts the number of cycles the system spends running MGS code. This is challenging because a large fraction of the overall MGS time is spent in handler code, invoked by MGS messages, that is interruptible. Figure 5.8 helps illustrate why counting cycles in interruptible handler code is difficult.

On the left half of Figure 5.8, the activity of a single Alewife processor is shown in time, where time progresses downward. Initially, before time $t0$, the processor is executing non-MGS code, indicated by a bar filled with a hash pattern. This code executes in context *C0*




```
handler work()
{
    unsigned begin, time;

    interrupts_off();
    begin = get_time();
    do_work();
    time = get_time() - begin;
}
```

Figure 5.8: Instrumenting timers in interruptible handlers.

of the processor, the default context for threads (recall from Section 5.2.2 that Sparcle, the Alewife integer core, supports four hardware contexts). At time $t0$, the thread is interrupted by an incoming message which invokes the handler code shown on the right half of Figure 5.8. As described in Section 5.2.2, Alewife runs the handler in the next available context, in this case context *C1*, to reduce the latency of message invocation.

Once running, the first thing the handler does is turn on interrupts[15]. Then, it performs some work which is timed by some statistics instrumentation code. The instrumentation code uses a routine called "get_time()" which returns the current value from a single hardware cycle counter.

In our example, before the handler running in context *C1* finishes, another message arrives at time $t1$ and invokes a new handler that get runs in context *C2* (for simplicity, the new handler runs the same code as the original handler). The new handler runs without interruption and completes at time $t2$, and execution returns to the original handler. Finally, the original handler completes at time $t3$, and execution returns to the background thread.

At the end of this sequence, the statistics code in the second handler returns $t2 - t1$, which is in fact the number of cycles the second handler runs on the processor. However, the statistics code in the first handler will return $t3 - t0$, which is incorrect because it counts the running time of both handlers. The result is that the overhead of the second handler is counted twice.

The solution to this problem is to use multiple cycle counters. Fortunately, Alewife provides four cycle counters in hardware; therefore, we dedicate one to each of the four hardware contexts in the processor. We modify the Alewife kernel to maintain the in-

[15]In Alewife, handlers begin running with interrupts off; this allows them to perform atomic operations. However, long running handlers must re-enable interrupts otherwise other incoming handlers are blocked until the running handler finishes. Blocked handlers remain in the network and can cause network congestion.

variant that only one cycle counter is ever turned on, and that counter is the one corresponding to the hardware context that is currently running. The other three counters are disabled so that they do not count. The invariant is maintained by instrumenting the message arrival and exit code in the kernel. On message arrival, the kernel disables whatever counter was on. Once disabled, the counter stops counting and holds its value until it is re-enabled (its value is not cleared!). The kernel then enables a new counter corresponding to the context in which the handler for the incoming message will run[16]. On message exit, the kernel disables this new counter, and re-enables the old counter before returning to the interrupted code.

By guaranteeing that the four hardware counters are non-overlapping, each counter will exactly track the number of cycles the processor spends running in each hardware context. This eliminates the problem of over counting handler overhead as illustrated in Figure 5.8.

5.3.6 User-Kernel Decomposition

In a production MGS system, all the software modules would be implemented inside the operating system kernel. In the implementation of our prototype, we place the software modules partly in kernel space, and partly in user space as libraries that are linked against the user's application. Our implementation strategy attempts to push as much of the MGS software into user space as possible. This strategy enhances testability, which is critical for the development of the system. In this section, we describe the decomposition of our MGS prototype into user and kernel modules, and we briefly discuss the implications such a decomposition has on performance.

Figure 5.9 shows the decomposition of our MGS prototype into user-space modules and kernel-space modules. The dotted line in the figure represents the separation between the user and kernel spaces. All shaded boxes represent software modules that belong explicitly to the MGS system. As the figure illustrates, most of the shared memory functionality is implemented at user level. The box labeled "MGS Library" which contains the Local-Client, Remote-Client, and Server Machines is implemented entirely in user space. And of course, the software address translation code is also implemented in user space since it is inlined by the compiler into the user's application.

The modules implemented in kernel space provide very simple functionality; therefore, it is less critical that they are not in user space from a software development standpoint. The TLB Fault Handler module intercepts TLB faults generated by the software address translation code in the application. This module performs an upcall into the MGS Library to invoke the Local-Client Machine; it is the mechanism by which control is passed from the application to MGS. The reason why this module lives in kernel space is explained below when we discuss performance. The Delayed Message module simulates the cost of inter-SSMP messages, as described in Section 5.3.2. The Local-Client, Remote-Client, and Server all interact with this module via system calls whenever a message is sent to a

[16] The kernel also clears this new counter so that we do not run the risk of overflowing the counter.

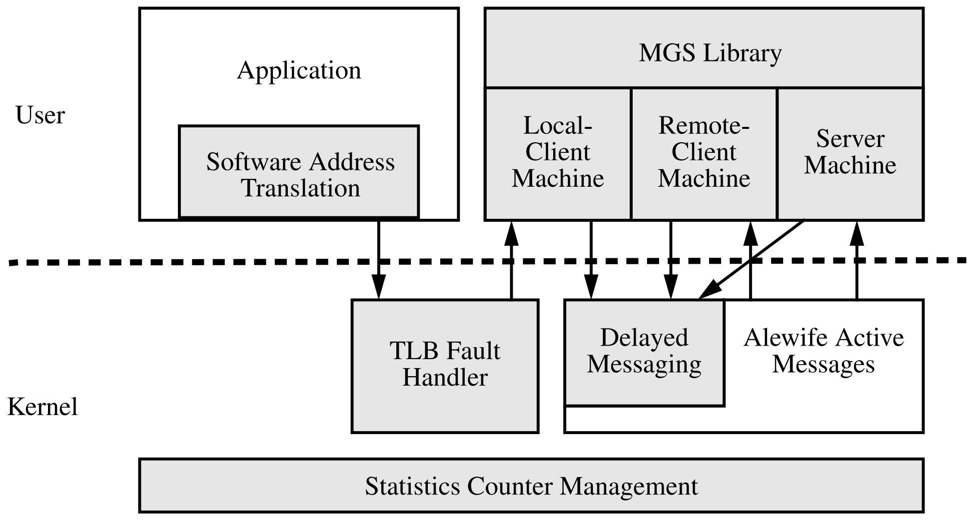


Figure 5.9: Decomposition of the MGS system into user-space and kernel-space modules.

protocol state machine on a remote SSMP. The Delayed Message module must reside in the kernel because it uses Alewife's timer facility which can only be accessed through the kernel. The Statistics Counter Management module manages the four Alewife hardware counters to ensure that they count in a non-overlapping fashion, as is required by the statistics instrumentation code described in Section 5.3.5. Since this module must be notified each time a new hardware context is active, it is simpler to implement it inside the kernel where scheduling of hardware contexts is performed rather than upcalling into MGS library code each time a scheduling change occurs. The Statistics Counter Management module spans the width of Figure 5.9 to signify that the hardware counter values are accessed by statistics instrumentation in both the application and the MGS library code. Finally, the last kernel module in Figure 5.9 is the Alewife Active Messages module, which is part of Alewife's fast inter-processor messaging facility described in Section 5.2.2. It provides the raw messaging layer used by the Delayed Messaging module to send messages between SSMPs once they have been artificially delayed. It also invokes the Remote-Client and Server Machines each time an active message is received from a remote SSMP.

Choosing an implementation that decomposes the software modules and places them in both the user and kernel spaces as indicated in Figure 5.9 rather than a kernel-only implementation impacts the performance of the prototype system. The impact is an overall increase in the number of user-kernel space crossings. This is easy to see by looking at Figure 5.9. If the box labeled “MGS Library” were placed in the kernel, as it would in a kernel-only implementation of MGS, then all the arrows between the Local-Client, Remote-Client, and Server Machines and the messaging interfaces provided within the kernel would disappear. Placing the protocol state machines in user space as we have done for the sake of debugging ease forces the system to incur a user-kernel space crossing each time a message is sent or received.

Although our goal was to place as much in user space as possible for software development purposes, we deliberately placed the TLB Fault module in the kernel to force a user-kernel space crossing on a TLB fault. We could have easily saved this user-kernel space crossing by directly calling the Local-Client from the inline code that detects TLB faults. While this would have been more efficient, it would have been overly optimistic in comparison to a production system in which the Local-Client would be implemented inside the kernel. In our prototype, there are two user-kernel space crossings for each TLB fault: one to intercept the TLB fault inside the kernel, and another to upcall into the Local-Client Machine. The return from the Local-Client Machine back to the application does not go through the kernel. Instead, the TLB Fault module in the kernel, after saving the processor's registers, artificially sets up a series of links on the user's stack that allows the Local-Client to restore the proper registers and return directly to the trapping code in the application. Two user-kernel space crossings for TLB fault handling was an explicit design goal as it is what we would expect in a kernel-only implementation of MGS (one crossing to get into the kernel where the Local-Client executes, and another crossing to get back to the application).

Chapter 6

Experimental Results

This chapter reports on extensive experimental experience with our prototype of the MGS architecture. Two fundamental questions are addressed that concern the effectiveness of DSSMPs as high-performance parallel processing architectures. First, *how effective are DSSMPs relative to other parallel architectures?* To address this question, we will compare the behavior of multigrain systems to monolithic shared memory systems that use all-software or all-hardware approaches. We will ascertain the benefit of providing fine-grained hardware-supported shared memory within SSMP nodes. The second question we will address is *what bottlenecks prevent DSSMPs from achieving higher performance?* We will look carefully at these bottlenecks to understand how they can be addressed through locality-enhancing program transformations. We will study the engineering effort required to implement such transformations, in addition to their impact on DSSMP performance.

With these two fundamental questions as our goal, we will first provide the context needed to address these questions. Section 6.1 presents micro-measurements that detail the cost of shared memory operations on our MGS prototype. These numbers provide a low-level characterization of system behavior before any applications are considered. In Section 6.2, we introduce a performance framework for characterizing the behavior of applications on DSSMPs. This framework facilitates a consistent and meaningful comparison of DSSMP performance against all-software and all-hardware shared memory performance. The framework will be used heavily throughout the rest of the chapter to present application results. Then, the two DSSMP performance questions mentioned above are addressed in Sections 6.3 and 6.4, respectively. Finally, Section 6.5 concludes the experimental results by examining the sensitivity of DSSMP performance to inter-SSMP messaging latency and system page size.

6.1 Micro-Measurements

In this section, we report measurements that characterize the cost of performing shared memory operations on the MGS prototype. Specifically, we consider three classes of overheads that relate to shared memory: overheads in cache-coherent shared memory,

Type	Home	Latency
Load	local	11
	remote	38
	remote (2-party)	42
	remote (3-party)	63
	remote software	425
Store	local	12
	remote	38
	remote (2-party)	43
	remote (3-party)	66
	remote software	707

Table 6.1: Cache-miss penalties on Alewife. All measurements are in cycles.

overheads for software address translation, and overheads in page-based software shared memory.

Table 6.1 reports overheads associated with cache-coherent shared memory by enumerating several types of cache-miss penalties. These numbers reflect shared memory performance provided on a single Alewife machine. These data appear in [23], and have been reprinted here.

Cache-miss penalties are reported for the two types of shared memory accesses, loads and stores, as indicated by the column labeled “Type” in Table 6.1. Since Alewife is a distributed-memory architecture (see discussion in Section 5.2 of Chapter 5), the miss penalty depends on whether the cache miss can be serviced locally or remotely. The column labeled “Home” differentiates between local and remote cache misses by specifying whether the home memory module for the cache line is “local” or “remote” to the node suffering the cache miss. Table 6.1 also shows the additional cost of invalidation during a cache miss by reporting the miss penalty when one or two outstanding cache copies must be invalidated, as indicated by the “2-party” and “3-party” remote penalties, respectively. Finally, the cost of software extension of the cache directory beyond 5 hardware pointers in Alewife (see discussion on LimitLESS in Section 5.2.1 of Chapter 5) is reported by the “remote software” latencies. In the case of a load, the miss penalty reported is the time required for a remote read miss to be serviced by a software handler. For stores, the miss penalty reported represents the latency seen by a write cache miss to a location with 6 outstanding read copies. This overhead includes the cost of sending 6 invalidation messages in a software handler, and receiving the acknowledgments in hardware.

All cache-miss penalties in Table 6.1 assume an unloaded machine, and therefore represent the maximum throughput attainable in the absence of contention in both the network and at memory modules.

Table 6.2 reports overheads associated with address translation in software virtual memory. As discussed in Section 5.3.1 of Chapter 5, MGS compensates for the lack of hardware support in Alewife for virtual memory by performing address translation in

SVM Operation	Latency
Mapping Check	6
Mapping Check and Translation	23
Distributed Array Translation	16

Table 6.2: Software Virtual Memory costs on MGS. All values are in cycles.

Type	Description	Latency	Serv Occ	Re-Cli Occ
Load	TLB Fault	2302		
	Page Fault	11772	2240	
	Page Fault, Single-Writer	29353	3557	6407
Store	TLB Fault	3590		
	Page Fault	21956	2400	
	Upgrade Fault	12441	150	
	Page Fault, Single-Writer	35293	3659	6373
Release	Single-Writer Transition	9992	2803	
	2-party Invalidation	33424	10086	11428
	3-party Invalidation	33516	17596	13015

Table 6.3: Software shared memory costs on MGS. All values are in cycles.

compiler-generated translation code inlined before each access made to a mapped (or potentially mapped) memory object.

There are two sources of software address translation overhead: checking code and translation code. Checking code is needed for memory references that potentially access mapped objects which could not be resolved at compile time. The checking code determines at run time whether the reference accesses a mapped or unmapped object. The cost of the checking code is given by the first row in Table 6.2. This cost is exactly the overhead incurred for accesses to "Local Heap Variables" (see Table 5.1) in which the checking code determines that a memory reference accesses an unmapped object, and thus the translation code can be bypassed. Translation code actually performs the address translation once a check determines that translation is necessary (*i.e.* a reference accesses a mapped object). The combined cost of the checking code and translation code is given by the second row. This corresponds to the cost for accesses to "Global Heap Variables." Finally, some memory references only incur the cost of translation because the compiler can determine statically that these references always access mapped objects. Accesses to distributed arrays fall into this category. The cost for a distributed array reference is given by the last row.

All the overheads reported in Table 6.2 were obtained by counting instructions and thus optimistically assume all inline code hit in the cache. Also, the overheads only account for inline code preceding an access and thus do not include the cost of the access itself.

The last set of micro-measurements, presented in Table 6.3, characterize the overheads seen in page-based software shared memory. Like Table 6.1, the overheads are grouped into different types of shared memory operations. In Table 6.3, there are three different types of shared memory operations, loads, stores, and releases, as indicated in the column marked "Type." The second column, marked "Description," specifies the action taken on a particular operation. For loads and stores, there are four different actions: "TLB Fault," "Page Fault," "Page Fault, Single-Writer," and "Upgrade Fault." A TLB fault occurs when a load or store is issued for which a local copy of the page containing the location exists in the local SSMP, but for which the processor issuing the load or store does not have a mapping. A page fault occurs when a load or store is issued for which no local copy of the page containing the location exists in the local SSMP. "Page Fault, Single-Writer" is similar to a page fault, except that the desired page faulted on is in Single-Writer mode (see Section 4.1.2 of Chapter 4) on a remote SSMP. An upgrade fault occurs when a store is issued for which the page containing the location is resident in the local SSMP, but the access privilege of that page is insufficient (*i.e.* the page is in read mode instead of write mode).

In addition to the four actions for loads and stores, there are three additional actions associated with releases reported in Table 6.3: "Single-Writer Transition," "2-party Invalidation," and "3-party Invalidation." A single-writer transition occurs when a release is performed on a page with a single outstanding write copy. In this case, the owner of the page is allowed to relax the consistency of the page past the release point (again, see Section 4.1.2 of Chapter 4). A 2-party (3-party) invalidation occurs when a release is performed on a page with two (three) outstanding copies resulting in the invalidation of both (all three) copies. In Table 6.3, all outstanding copies are in write mode, each with modifications performed to half the page.

The last three columns in Table 6.3 report overheads associated with each action described above. The first of the three columns, labeled "Latency," reports the latency seen by the requesting processor. This is the number of cycles the requesting processor is stalled as the software shared memory operation is performed. The next column, labeled "Serv Occ," reports the server occupancy. This is the total number of cycles spent executing the Server Machine for the page in question. These cycles are incurred by the processor responsible for executing the Server Machine in the Server SSMP. TLB faults do not incur server occupancy because they do not invoke the Server Machine. Finally, the last column, labeled "Re-Cli Occ," reports the remote client occupancy. This is the number of cycles spent executing the Remote Client Machine in order to perform invalidation. These cycles are incurred on the processors that own outstanding pages being invalidated. Only those actions that involve invalidation incur remote client occupancy. Both occupancy measurements account for the cost of executing the handlers, but do not include the cost of the interrupt, dispatch code, and return code necessary to invoke and exit from the handler (this cost is approximately 200 cycles on Alewife).

All values in Table 6.3 are averages over several repetitions of each operation. Except for the instrumentation code that repeatedly executes each operation and measures their cost, the MGS prototype used for the micro-measurements is otherwise idle; therefore, the numbers in Table 6.3 represent the highest throughput attainable in the absence of contention both in the network and at all processors involved. Also, we assume a

1000 cycle (50 μsec) latency for all inter-SSMP messages (*i.e.* the fixed delay parameter discussed in Section 5.3.2 of Chapter 5 is configured for 1000 cycles), and a page size of 1K-bytes. The inter-SSMP communication latency is fairly aggressive, but achievable on existing networks. The delay impacts the latency numbers, but does not affect the occupancy numbers in Table 6.3 (see Section 6.5 for a discussion on the impact of inter-SSMP communication latency and page size on system behavior).

There are two important remarks to be made regarding Table 6.3. First, a comparison between the latency columns of Tables 6.1 and 6.3 reveals that page-based software shared memory is roughly 3 orders of magnitude more expensive than cache-coherent shared memory on Alewife. This underscores the importance of optimizations that leverage cache-coherent shared memory whenever possible. Second, the Single-Writer mechanism has the potential to provide large savings in overhead as evidenced by the large latencies for releases that involve invalidation. A page that meets the single-writer condition will incur the penalties of the "Single-Writer Transition" entry on its first release. Not only does this have much less associated latency as compared with the "2-party" and "3-party" releases, but it also inflicts much less occupancy overhead. Furthermore, once a page transitions into the single-writer mode, it incurs 0 overhead on subsequent releases until it transitions out of single-writer mode, caused by a page fault from a remote SSMP (corresponding to the "Page Fault, Single-Writer" entry).

It is also interesting to note that faults due to stores are more expensive than faults due to loads. This is because a page with write access privilege requires a twin copy to be made. Also, the latency of releases that require invalidation (in an unloaded system) is generally insensitive to the number of outstanding copies as evidenced by the similar latencies for 2-party and 3-party releases. This is because the invalidation requests are sent by the Server Machine simultaneously; thus the invalidations happen in parallel. However, the occupancy overhead increases with wider sharing since the occupancy numbers reported in the "Re-Cli Occ" column are incurred once for each copy invalidated.

6.2 Performance Framework

In this section, we introduce a performance framework that enables a crisp characterization of application performance on DSSMPs. The framework is used extensively later on in Sections 6.3 and 6.4 to present results from our application study.

Our performance framework is based on two key system parameters that describe a DSSMP configuration: the total number of processors, P, and the number of processors in each SSMP, or SSMP node size, C. The performance framework characterizes the behavior of an application in the following way. Given a DSSMP with a fixed total machine size P, we measure an application's performance on the DSSMP as the SSMP node size C is varied from 1 to P. We call this set of measurements the application's *performance profile*.

The performance profile tells us how an application responds to a change in the mix-

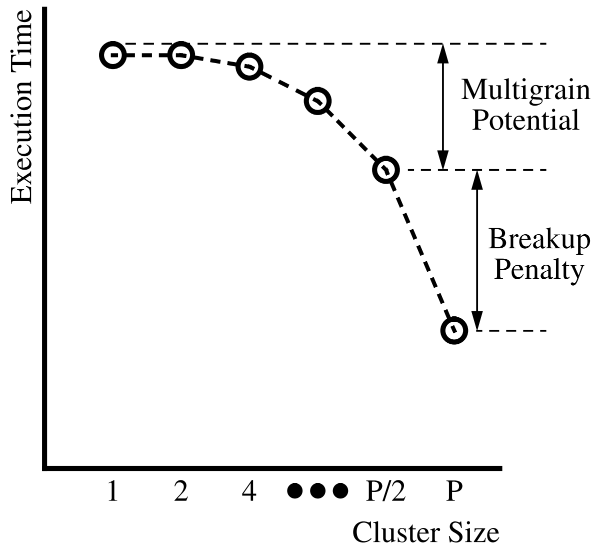


Figure 6.1: A hypothetical application analyzed using the performance framework. This application is not well-suited for DSSMPs.

ture of hardware and software in the implementation of multigrain shared memory. A large SSMP node size implies a greater degree of hardware shared memory support across the DSSMP; consequently, a greater fraction of the application's shared memory accesses will be satisfied in hardware, and most sharing will occur at cache-line granularity. Conversely, a small SSMP node size implies the DSSMP relies more on software shared memory support; consequently, a greater fraction of the application's shared memory accesses will be satisfied in software, and more sharing will occur at page granularity.

Both endpoints of the performance profile, where SSMP node size is 1 and P processors respectively, are interesting because each represents a collapse of the DSSMP network hierarchy. At $C = 1$, each SSMP is a uniprocessor, so there is no internal network. This means that all shared memory accesses to remote locations use software and share at page granularity. Conversely, at $C = P$, there is only one SSMP, and it is the entire system. There is no external network. All remote accesses are handled in hardware and share at cache-line granularity. Therefore, the endpoints of the performance profile allow us to compare the performance of intermediate DSSMP configurations against the degenerate all-software or all-hardware shared memory architectures[1].

Figure 6.1 shows the performance profile of a hypothetical application. Execution time is plotted against the SSMP node size parameter, C, in powers of 2 for a total system

[1]The performance at an SSMP node size of 1 and P processors does not correspond exactly to an all-software and all-hardware shared memory system, respectively, given our MGS prototype. This is because MGS provides functionality that is necessary in a DSSMP, but would not be necessary in either a network of uniprocessor workstations (all-software DSM) or an MPP (all-hardware DSM). When we present experimental data in Section 6.3, we substitute the native Alewife performance numbers (without MGS) for the MPP configuration, but we make no attempt to correct the all-software DSM performance numbers.

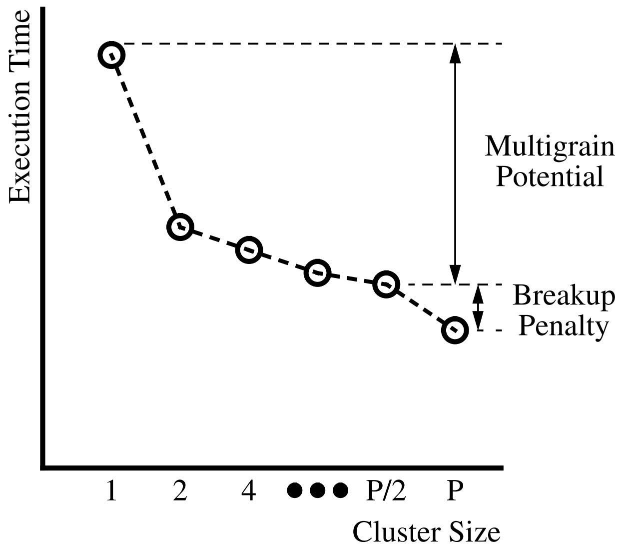


Figure 6.2: A hypothetical application analyzed using the performance framework. This application is well-suited for DSSMPs.

size, P. While the performance profile visually conveys the application's sensitivity to SSMP node size, we define two quantitative metrics that identify the most important features on the performance profile, and thus can be used to characterize application behavior. These metrics have been labeled in Figure 6.1, and are:

Breakup Penalty. The execution time increase between the P SSMP node size and the $\frac{P}{2}$ SSMP node size is called the "breakup penalty." This is the minimum performance penalty incurred by breaking a tightly-coupled (all-hardware shared memory) machine into a clustered machine.

Multigrain Potential. The difference in execution time between an SSMP node size of 1 and an SSMP node size of $\frac{P}{2}$ is called the "multigrain potential." The multigrain potential measures the performance benefit derived by capturing fine-grain sharing within SSMP nodes.

A third feature of the performance profile in Figure 6.1 that is important for characterizing an application's behavior on DSSMPs, but for which we do not explicitly define a metric, is the curvature of the performance profile across the multigrain potential. A concave curvature indicates most of the multigrain potential is achieved at large SSMP node sizes, while a convex curvature indicates most of the multigrain potential is achieved at small SSMP node sizes. The curvature is important because it determines whether DSSMPs built using small SSMP nodes can be effective, or whether large SSMP nodes are required for performance. While we do not measure the curvature quantitatively, we will refer to the performance profile's curvature and the impact of using small versus large SSMP nodes throughout the rest of this thesis.

Our performance framework tells us that the hypothetical application in Figure 6.1 is not well-suited for DSSMPs. First, the application's performance profile has a large breakup penalty. This indicates that the application will perform poorly on the DSSMP as compared to an all-hardware cache-coherent DSM. Second, the multigrain potential is small indicating that very little benefit is derived from the hardware-supported shared memory provided within SSMP nodes; therefore, this application will not achieve much higher performance on a DSSMP as compared to an all-software DSM. Finally, the curvature of the performance profile across the multigrain potential is concave indicating that what little multigrain potential there is can only be realized if the DSSMP consists of a few very large SSMPs.

In contrast, Figure 6.2 shows the analysis of another hypothetical application, again using our performance framework. The performance profile presented in Figure 6.2 displays a very small breakup penalty. This application will do almost as well on a DSSMP as it will on an all-hardware system because there is very little loss in performance due to introducing software in the shared memory implementation. The performance profile has a large multigrain potential indicating large benefits derived from capturing fine-grain sharing in SSMP nodes. And the curvature of the performance profile across the multigrain potential is convex with a steep slope at small SSMP node sizes. This indicates that most of the multigrain potential can be achieved at small SSMP node sizes. The implication for the application depicted in Figure 6.2 is that it will perform well on DSSMPs constructed from small-scale multiprocessors.

As we will see in Section 6.3.2, many of the applications from our application suite have challenging fine-grain communications requirements. DSSMPs deliver decent performance on some of these challenging applications; however, breakup penalties are significant because the fine-grain communication patterns uniformly span the entire machine resulting in a performance profile that resembles Figure 6.1. Section 6.4 will show that locality-enhancing transformations can be applied in order to cluster the fine-grain sharing patterns found in these challenging applications. Most of the transformations are simple and resemble the transformations performed by existing parallel optimizing compilers. On the transformed applications with improved locality characteristics, we see performance profiles that resemble 6.2.

6.3 Applications

While Section 6.1 presents detailed measurements on shared memory operations, such system-level performance numbers do not capture overall system behavior. In this section, we study the behavior of applications on the MGS prototype so that we may characterize end-to-end performance.

The primary intent of this section is to understand how the MGS system behaves on *off-the-shelf* applications, *i.e.* applications that have been written only with a generic parallel shared memory machine model in mind. These applications are not aware of the underlying multigrain support for shared memory nor the clustered nature of the DSSMP

Application	Problem Size	Lines
Jacobi	1024 × 1024 Grid, 10 Iterations	205
Matmul	256 × 256 Matrices	239
FFT	32K Elements	322
Gauss	512 × 512 Matrix	322
Water	343 Molecules, 2 Iterations	2090
Water-Kernel	512 Molecules, 1 Iteration	
Water-Kernel-NS	512 Molecules, 1 Iteration	
Barnes-Hut	2K Bodies, 3 Iterations	4058
TSP	10-City Tour	665
Unstructured	2800 Nodes, 17377 Edges, 1 Iteration	9094
Unstructured-Kernel	2800 Nodes, 17377 Edges, 1 Iteration	

Table 6.4: List of applications, their problem sizes, and their size in number of lines of C code.

upon which they run[2]. While we are very interested in the performance improvements that can be gained by exposing details of the shared memory layer to the application, we defer these issues to Section 6.4. Therefore, the results presented in this section represent the performance that DSSMPs can deliver with minimal effort from either the programmer or the compiler.

First, we describe the applications used in our study in Section 6.3.1, and then we present detailed experimental results in Section 6.3.2.

6.3.1 Application Suite

Table 6.4 lists the applications used in this thesis to study the overall system behavior of DSSMPs. There are eight applications in total, two of which (Water and Unstructured) have variants which facilitate more detailed study later in Section 6.4 and Chapter 7. While all of the applications are scientific in nature, there is a mixture of codes that have both regular and irregular memory access patterns, and both static and dynamic control flow behavior. The table includes the problem size used for our experiments, and the number of lines of C code in the application (the number of lines of C code have been omitted for Water-Kernel, Water-Kernel-NS, and Unstructured-Kernel as these are variants on the main Water and Unstructured applications).

The first four applications, all exhibit regular memory access patterns with static control flow. Jacobi performs an iterative relaxation over a two-dimensional grid, while Matmul multiplies two dense matrices. FFT computes a one-dimensional fast Fourier transform, and Gauss performs Gaussian elimination on a matrix. Water is an application from the SPLASH-I benchmark suite [61]. It is a molecular dynamics code that

[2]We do, however, allow applications to use the multigrain-aware synchronization library described in Section 4.3 of Chapter 4. But this only requires linking the application against our library, and thus, we do not count this as additional programmer effort.

Application	Seq	Sp1	SVM-Seq	Ovhd	SVM-Par	Sp2
Jacobi*	1020816028	28.3	1618916600	1.59	53889697	30.0
Matmul	1967397265	31.3	3080884002	1.57	114667516	26.9
FFT	495224878	13.6	491769198	0.99	41487459	11.9
Gauss	2666915900	15.9	5034851631	1.89	217332821	23.2
Water	1284906732	26.1	1960029691	1.53	72948004	26.9
Water-Kernel			1532197479		58465483	26.2
Water-Kernel-NS			2122687515		75058889	28.3
Barnes-Hut	563916197	13.4	976160390	1.73	72772466	13.4
TSP	27371714	8.0	53485523	1.95	3040273	17.6
Unstructured	371716843	17.4	1260702520	3.39	87473784	14.4
Unstructured-Kernel			204001329		13444073	15.2

Table 6.5: Baseline application performance. "Seq" and "Sp1" report sequential running time and speedup, respectively, on an Alewife machine without SVM. "SVM-Seq" reports sequential running time with SVM. "Ovhd" is the amount of SVM overhead. "SVM-Par" reports running time with SVM on a 32-node Alewife machine, "Sp2" reports speedup with SVM on 32 nodes.

simulates the motion of water molecules in three-dimensional space. Again, this application has fairly regular memory access patterns with static control flow. Water-Kernel and Water-Kernel-NS are variants on the basic Water application, and are explained in Section 6.4 and Section 7.2.1 of Chapter 7, respectively. Barnes-Hut is a hierarchical N-body simulation, also from the SPLASH-I suite. The algorithm uses an octree data structure to sort the bodies according to their positions in space. The octree logarhythmically reduces the number of body interactions by enabling each body to interact with the summary of a progressively larger number of bodies as interaction distance increases. Because the structure of the octree is highly data dependent, the memory access patterns are irregular and control flow is dynamic in Barnes-Hut. TSP is the traveling salesman problem that uses a branch and bound algorithm and a centralized work queue to distribute work. Because of the pruning inherent to branch and bound algorithms, TSP has dynamic (data-dependent) control flow. Finally, Unstructured is a computation over an unstructured mesh from the University of Wisconsin, Madison, and the University of Maryland, College Park [52]. The computation resembles solving Eular equations on unstructured meshes, but does not actually produce meaningful numeric results. The code exhibits highly irregular memory access patterns and dynamic control flow. Unstructured-Kernel is a variant of Unstructured, and is explained in Section 6.4.

Table 6.5 provides baseline performance numbers for our applications on Alewife without the overheads of software shared memory that would be incurred in a DSSMP. The first two columns report performance numbers on Alewife without any software address translation overhead, *i.e.* native Alewife performance. The "Seq" column reports running time on a single-node Alewife machine (we do not report "Seq" numbers for the Water and Unstructured variants because they are similar to the original versions of the applications), and the "Sp1" column reports the speedup on a 32-node Alewife machine. In the case of Jacobi, the "*" symbol signifies that its problem size was not able to fit in the memory of a single Alewife node; therefore, for Jacobi, we ran the problem on 4 nodes for both the "Seq" and "SVM-Seq" (explained below) columns, and extrapolated the single node numbers by assuming linear speedup from 1 to 4 processors.

The last four columns report baseline performance for the applications with software virtual memory, *i.e.* these numbers include the software address translation overheads described in Section 5.3.1 of Chapter 5. "SVM-Seq" reports single-node performance on Alewife with software virtual memory. The next column, labeled "Ovhd," is the ratio of the "SVM-Seq" and "Seq" columns. This is the dilation in sequential running time due to software address translation, and thus quantifies the cost of software virtual memory. Notice that SVM overhead is highly application dependent. While there is no detectable dilation in FFT, most applications become 50% – 100% slower due to SVM overhead. In the extreme case, Unstructured is over three times slower with SVM than without SVM. This is because Unstructured spends all of its time in several tight loops, each accessing mapped memory objects with very little computation between accesses. We discuss the expected impact of software address translation overhead on our results below.

The column labeled "SVM-Par" reports the running time on a 32-node Alewife machine. These parallel performance numbers include the overhead of software address translation, but do not include any other MGS-related overheads. In particular, the system initializes all mappings needed by the application to write mode before the application begins execution, so the application never suffers TLB faults or page faults. Furthermore, instead of using the multigrain synchronization primitives described in Section 4.3 of Chapter 4, we use standard shared memory synchronization primitives, as provided by the P4 macro library[3] [12]. Therefore, the performance reported in the "SVM-Par" column is the performance on a hardware cache-coherent DSM (modulo software address translation), and is what we compare DSSMP performance against later in Section 6.3.2.

Finally, the last column in Table 6.5, labeled "Sp2," is the speedup attained on 32 nodes with software address translation (the ratio of the "SVM-Seq" and "SVM-Par" columns). Except for the Jacobi application, an application known for its excellent speedup, all our applications exhibit only modest to good speedups. This indicates that the introduction of SVM overhead, which we expect to parallelize perfectly, does not increase the computation-to-communication ratio of our applications to the point that they become embarrassingly parallel. Instead, even with SVM overhead, it is still challenging to achieve high speedups on our applications.

[3] The P4 primitives are cheaper than the multigrain primitives because they don't include the optimizations for clustering.

Impact of Software Address Translation Overhead

Inlining code to perform software address translation significantly slows down our applications. As indicated by the "Ovhd" column in Table 6.5, most of our applications show a dilation in sequential running time between 50% – 100%. Notice that this added overhead parallelizes perfectly since the inlined code does not perform communication. Therefore, by introducing software virtual memory, we necessarily increase the computation-to-communication ratio of the applications. In other words, software virtual memory makes it easier to achieve good parallel performance.

How do we assess the impact of software address translation? One way to view software address translation is that it takes a shared memory application and creates a "new" shared memory application (call it application') with different communication and computation requirements. Therefore, the way to interpret the results presented in the rest of this chapter is to recognize that they represent *exactly* the performance one would expect on a DSSMP for application'. Our goal in this discussion is to show that application' for all the applications we study do not become embarrassingly parallel[4] due to the increase in the computation-to-communication ratio caused by software address translation code.

Towards this goal, we make two observations. First, except for Jacobi, all of the applications in our suite are not embarrassingly parallel even after SVM instrumentation. As the column labeled "Sp2" in Table 6.5 indicates, Matmul, Gauss, and Water achieve good speedups, but not linear speedups, and FFT, Barnes-Hut, TSP, and Unstructured all have speedups that are close to or below 16, which represents only a 50% efficiency since the speedups are measured on a 32-node machine. We conclude that our application suite presents challenging communication requirements despite the increase in the computation-to-communication ratio due to SVM instrumentation.

Second, and somewhat surprisingly, we observe that SVM instrumentation does not significantly increase parallel performance for most of the applications. By comparing the columns "Sp2" and "Sp1" of Table 6.5, we can observe the impact on speedup due to SVM instrumentation. We find that for Jacobi, Water, and Barnes-Hut, SVM instrumentation improves speedup by only 6% or less. For Matmul, FFT, and Unstructured, speedup actually gets worse after SVM instrumentation. While we do not have a definitive explanation for this, we speculate that the benefits of increased computation-to-communication ratio are compensated by a decrease in cache performance due to code expansion caused by the inline SVM code. The only applications that exhibit significant improvements in speedup due to SVM instrumentation are Gauss and TSP. Gauss experiences a 46% increase in speedup, and TSP experiences a 120% increase in speedup. We do not expect this to significantly impact our results. As we will see in Section 6.3.2, Gauss is compute bound and achieves good performance on DSSMPs. The version of Gauss without SVM instrumentation, which would have 46% less parallelizable overhead, would still be compute bound; therefore, we expect the same conclusion for Gauss. As

[4] By embarrassingly parallel application, we mean an application that achieves close to linear speedup.

Section 6.3.2 will also show, TSP is extremely fine-grained even with SVM instrumentation. Our conclusion for TSP is that it does not run well on DSSMPs. The version of TSP without SVM instrumentation would be even finer-grained and thus exhibit even worse performance on DSSMPs. This does not change our conclusion for TSP.

6.3.2 Application Results

This section presents detailed experimental results of the applications listed in Table 6.4, excluding the kernels (Water-Kernel, Water-Kernel-NS, and Unstructured-Kernel). These kernels are variants on the original Water and Unstructured applications and will be studied in Section 6.4 and Chapter 7. All measurements were performed on our MGS prototype, running on a 32-node 20 MHz Alewife machine. The inter-SSMP communication latency used is 1000 cycles (50 μsec), and the page size is 1K-bytes. Section 6.5 later examines the impact of varying communication latency and page size on performance.

The results for the individual applications appear in Figures 6.3 through 6.12. We present the data using the performance framework discussed in Section 6.2. For each application, we observe the application's execution time (y-axis) on a 32-processor DSSMP as SSMP node size is varied from 1 to 32 in powers of 2 (x-axis). It is important to emphasize that all data points reported in Figures 6.3 through 6.12 were measured on a 32-processor machine; the only parameter being varied is SSMP node size, and thus, also the number of SSMP nodes comprising the DSSMP.

Each execution time data point in Figures 6.3 through 6.12 have been broken down into four components: time spent in user code, time spent in synchronization (for both locks and barriers), and time spent in the MGS runtime layer. The four components are labeled "User," "Lock," "Barrier," and "MGS," respectively. The user component not only counts useful cycles in user code, but it also counts cycles spent in software address translation and Alewife cache-coherent shared memory stall time. The synchronization components include both the overhead of executing synchronization code and waiting on synchronization conditions.

The 32-processor SSMP node size data points (the rightmost bars in Figures 6.3 through 6.12) are exactly the runtimes reported in the "SVM-Par" column of Table 6.5. For these runs, the system initializes all mappings needed by the application to write mode before the application begins execution, so there are no cold misses associated with the mapping state. Also, a 32-processor SSMP node size means that the DSSMP consists of a single SSMP, so there is no inter-SSMP coherence traffic. Therefore, the MGS component for these runs is zero. Furthermore, instead of using the multigrain synchronization primitives, these executions use the synchronization primitives provided by the P4 macro library. The cost of synchronization in the P4 library has been folded into the user component because we did not instrument cycle counting in the P4 library. The 32-processor data points represent the performance of the applications on a tightly-coupled MPP that has hardware-supported cache-coherent shared memory.

Table 6.6 summarizes the experimental results that will be discussed in detail in the rest of this section. The first two columns of data report the Multigrain Potential and

Application	MP	BP	S1	S2	S4	S8	S16
Easy Category							
Jacobi	3	-2	29.7	30.5	30.5	30.7	30.7
Matmul	-6	1	28.2	26.7	26.8	26.8	26.5
FFT	10	1	10.7	11.2	11.5	11.7	11.8
Gauss	-7	3	24.2	26.5	27.3	26.0	22.5
Challenging Category							
Water	82	159	5.7	6.4	8.1	10.0	10.4
Barnes-Hut	61	193	2.8	3.0	3.3	3.6	4.6
Pathologic Category							
TSP	80	1014	0.9	1.0	1.2	1.5	1.6
Unstructured	88	641	1.0	1.2	1.5	1.6	1.9

Table 6.6: Summary of application performance on DSSMPs. The "MP" column reports Multigrain Potential, and the "BP" column reports Breakup Penalty[6]. The last five columns report speedups with SSMP nodes of size 1–16, in powers of two, on a machine with 32 total processors.

the Breakup Penalty as defined in our performance framework; these data also appear alongside the graphs presented in Figures 6.3 through 6.12. In addition, the last five columns of data report speedups obtained on DSSMPs with SSMP nodes of size 1 through 16, in powers of two. These speedup results will be referenced when the experimental results are discussed.

Based on the performance of the applications reflected in the results, we identify three categories: easy, challenging, and pathologic. We present detailed explanations of the results below organized using this taxonomy.

Easy Category

Jacobi, Matmul, FFT, and Gauss, presented in Figures 6.3, 6.4, 6.5, and 6.6, respectively, belong to the "easy" category. These applications have in common a small multigrain potential and a small breakup penalty. The small multigrain potential indicates that very little benefit is experienced as SSMP node size is increased and more hardware cache-coherent shared memory is provided in each SSMP node. The small breakup penalty indicates that DSSMPs closely match the performance of MPPs on these applications. The combination of a small multigrain potential and a small breakup penalty implies that the performance profile for applications in the easy category is *flat*.

A flat performance profile signifies that the application is insensitive to the particular implementation of the shared memory layer provided underneath the application.

[6]The negative multigrain potentials are due to the preference that LimitLESS gives to smaller SSMP node sizes, and the negative breakup penalty is due to the benefits of bulk data movement provided by page-level replication. These anomalous effects are described in the detailed discussion of the application results.

Whether shared memory is supported in software at page granularity or in hardware at cache-line granularity, the application will perform well regardless. Therefore, for these applications, DSSMPs deliver good performance, but they do not provide any performance benefit over traditional software DSM or hardware MPP architectures.

To understand why applications in the easy category are insensitive to shared memory implementation requires a closer look at their sharing patterns. The four applications in this category exhibit coarse-grained sharing: each processor performs large amounts of independent work before communicating with other processors. For instance, Jacobi is an iterative algorithm in which new values are produced each iteration based on computation over a dense 2-D matrix. However, the values produced by a processor in an iteration are not needed by other processors until the next iteration. Moreover, only a fraction of the produced values (which grows as the square-root of the number of produced values) need to be communicated, so the communication volume is small compared with the amount of computation. In Matmul, the entire computation proceeds without any communication between processors. Each processor reads specific rows and columns of two input matrices, so there is some movement of data. But the values produced by each processor are never consumed by other processors. FFT is similar to Jacobi in that the algorithm proceeds iteratively with values communicated only across iterations. There is, however, more communication relative to the amount of computation in FFT (communication volume is linear with the amount of computation). Also, there is significant load imbalance in our implementation since the amount of work performed by each processor is not equal, as evidenced by a large barrier overhead in Figure 6.5. Finally, in Gauss, processors are responsible for computing values for a set of rows in a large matrix, but the values are communicated only at the end of a pass over the entire matrix and only by one processor, the processor owning the current pivot row. An array of locks is used to signal when a new pivot row has been completed, so there is noticeable lock overhead in Figure 6.6. In general, coarse-grained sharing exhibited by the four applications in the easy category leads to a very high computation-to-communication ratio as evidenced by the lack of MGS overhead in Figures 6.3 through 6.6.

Coarse-grained sharing can be supported efficiently by any shared memory implementation because the communication happens infrequently, so the cost of each communication has little impact on end performance. Therefore, supporting communication in either hardware or software is equally adequate. In fact, it is possible for software mechanisms to outperform hardware mechanisms. This is the case for Jacobi, in which the breakup penalty is negative (*i.e.* the DSSMP is outperforming the MPP). The data communicated in Jacobi is densely packed. The MGS software shared memory layer transfers such dense data efficiently by using Alewife's DMA facility to move data in bulk messages. Once the data has been transferred, the processor needing the data can access it by suffering cache misses to local memory. In contrast, the all-hardware system uses cache-coherent shared memory for communication. Consequently, bulk data is moved by suffering a remote cache miss for every cache line in the bulk region. In general, negative breakup penalties are rare.

The negative multigrain potential exhibited by Gauss in Figure 6.6 is another anoma-

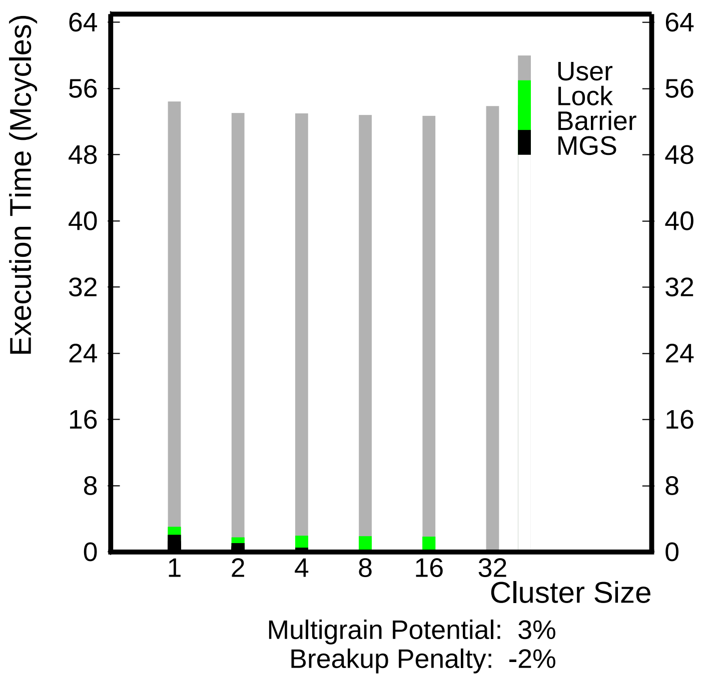


Figure 6.3: Results for Jacobi.

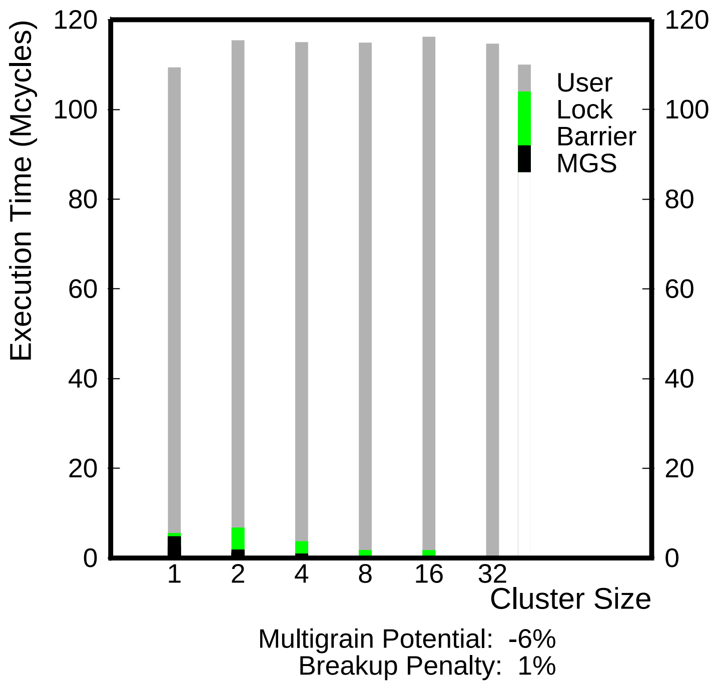


Figure 6.4: Results for Matrix Multiply.

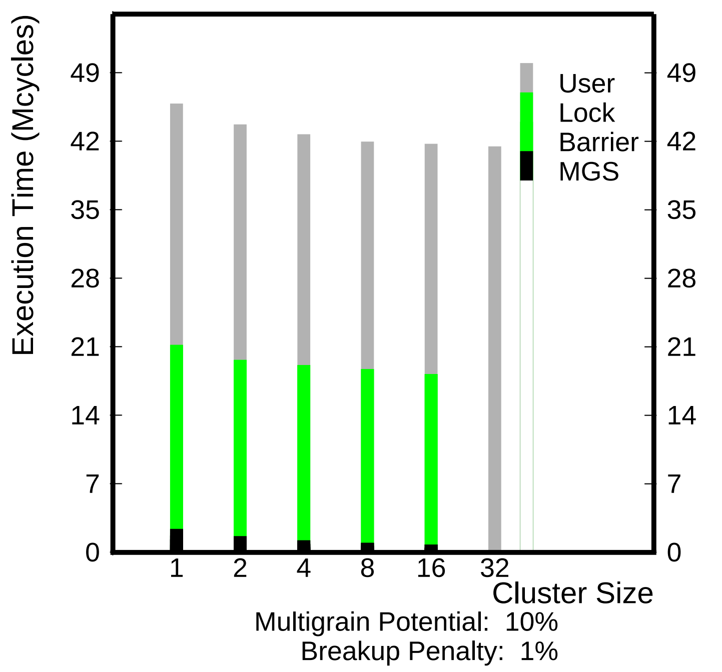


Figure 6.5: Results for FFT.

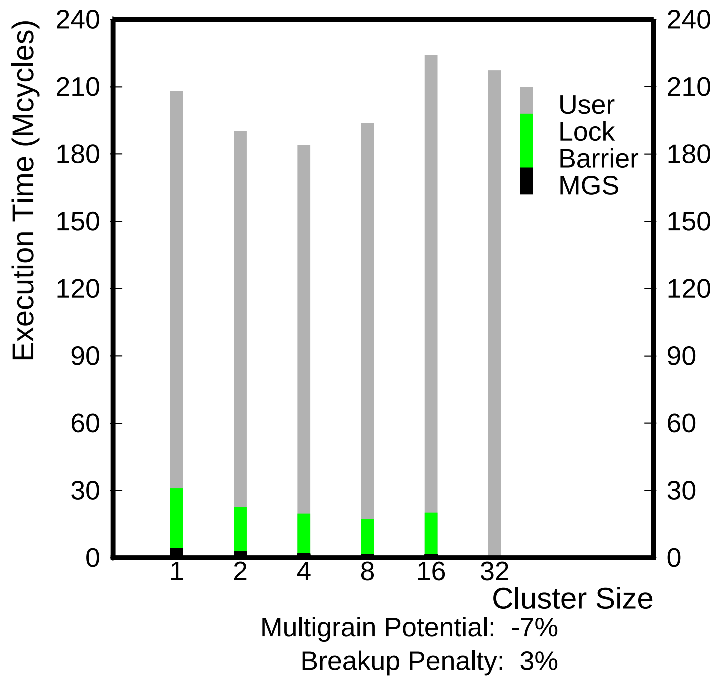


Figure 6.6: Results for Gauss.

lous condition. This is due to LimitLESS overhead as is discussed in Section 5.2.1 of Chapter 5. In Gauss, each pivot row is read by all processors in the system. Since Alewife only supports 5 sharers in its hardware directory, LimitLESS overhead is incurred when SSMP node size is increased from 4 processors to 8 processors. Because the cost of LimitLESS is quite high, as documented in Table 6.1, DSSMPs with smaller SSMP nodes outperform those with larger SSMP nodes. This effect would not appear in an SSMP that supports cache coherence fully in hardware (*i.e.* without software extension as is provided in Alewife).

Challenging Category

Water and Barnes-Hut, presented in Figures 6.7 and 6.8, respectively, belong to the "challenging" category. In this category, applications exhibit a performance profile that is far from flat, as was the case for applications in the easy category. Applications in the challenging category have both a large multigrain potential and a large breakup penalty. The large multigrain potential (82% for Water and 76% for Barnes-Hut) is a positive result for DSSMPs because it means that supplying hardware-supported cache-coherent shared memory between more processors (*i.e.* building larger SSMP nodes) improves performance. This suggests that DSSMPs offer better scalability than systems that only provide software support for shared memory. Unfortunately, the large breakup penalty (159% for Water and 231% for Barnes-Hut) is a negative result because it implies that there is a significant performance gap between DSSMPs and all-hardware shared memory systems; therefore, on these applications, MPPs hold a performance advantage over DSSMPs. To explain these results, we take a close look at the applications below.

As Figure 6.7 shows, the primary obstacle to higher performance in Water is the MGS component. The Water workload generates a significant amount of software shared memory traffic due to poor data locality. The software shared memory traffic invokes handlers in the MGS layer that appears as MGS overhead. While the synchronization overheads are significant as well, they are caused by the same effects which occur in the Barnes-Hut workload; therefore, we will address synchronization components when we discuss Barnes-Hut, where synchronization overhead is more pronounced.

Poor data locality occurs in the force interaction computation, an $O(N^2)$ computation where Water spends most of its execution time. The pseudo-C code for this computation appears in Figure 6.9. In this code example, N is the total number of molecules in the simulation, P is the total number of processors, pid is the processor ID of an individual processor, mol is the global array of molecule records, and s is the global array of locks, one for each molecule. The computation consists of a doubly-nested loop that iterates over pairings of molecules; the combined iteration spaces of the P processors considers all possible N^2 pairings over the N molecules. The loop body performs some computation based on the molecules indexed by i and j, and then atomically updates each of the two molecules using the locks in array s. The *release* operation in each atomic update ensures that the update is made visible to all other processors before the lock is relinquished.

There is significant temporal and spatial reuse of data in this loop; however, write

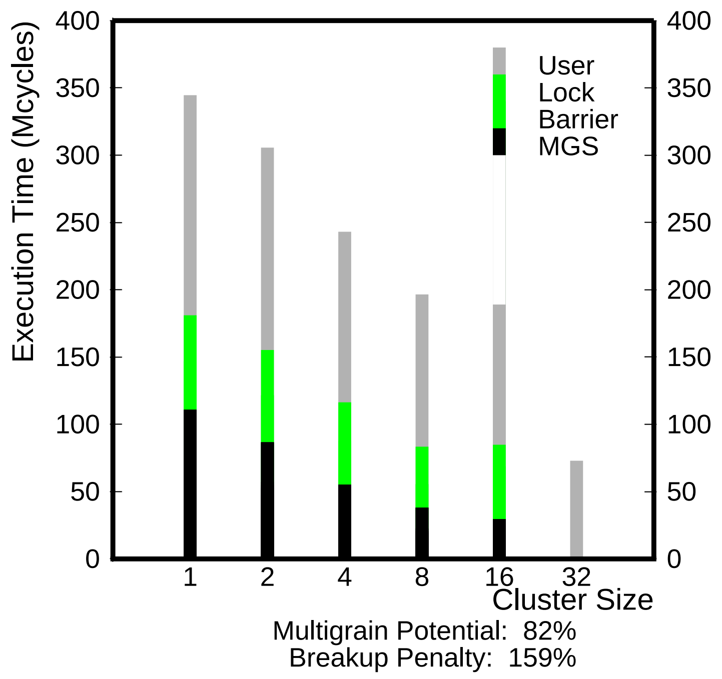


Figure 6.7: Results for Water.

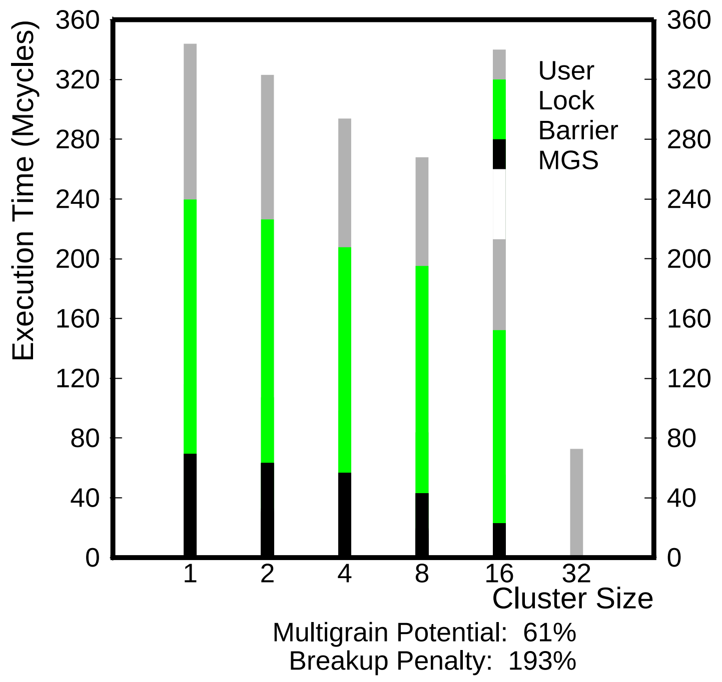


Figure 6.8: Results for Barnes-Hut.

```
for (i = (N/P)*pid; i < (N/P)*(pid+1); i++) {
    for (j = i+1; j < i + N/2; j++) {

        compute(mol[i%N], mol[j%N], &ai, &aj);

        lock(s[i%N]);
        mol[i%N] += ai;
        release();
        unlock(s[i%N]);

        lock(s[j%N]);
        mol[j%N] += aj;
        release();
        unlock(s[j%N]);
    }
}
```

Figure 6.9: Pseudo-C code for the force interaction computation in Water.

sharing prevents caching from fully capitalizing on such reuse. Write sharing occurs because there is significant overlap between the iteration spaces of different processors. While the outer loops on different processors produce indices that are disjoint along the i dimension, there is significant overlap between an inner loop against another inner loop and an inner loop against an outer loop on two separate processors.

The MGS overhead in Figure 6.7 is the result of invalidations due to write sharing between SSMPs. Notice that the impact of write sharing decreases as SSMP node size is increased. Larger SSMP nodes alleviate page-level invalidations by supporting a larger fraction of the write sharing in cache-coherent shared memory. Fine-grain support handles write sharing more effectively since false sharing is minimized by the smaller cache-line block size, and lower hardware latencies result in less processor stall time when write conflicts do occur. However, write sharing between SSMPs is still a problem even when SSMP node size is large because the memory accesses performed by the loop in Figure 6.9 are uniformly distributed across the range of memory locations occupied by the *mol* array. The global nature of the memory access patterns cause write invalidations at the page level regardless of the clustering configuration.

The results for Barnes-Hut appear in Figure 6.8. As the figure shows, the most significant source of slowdown in Barnes-Hut is lock overhead. Barrier overhead and MGS overhead are significant as well, though not as severe.

Barnes-Hut is a discrete-time simulation of N-body motion in 3-dimensional space. Instead of considering all N^2 possible interactions between bodies as is done in the Water workload, Barnes-Hut performs a significantly smaller number of interactions by

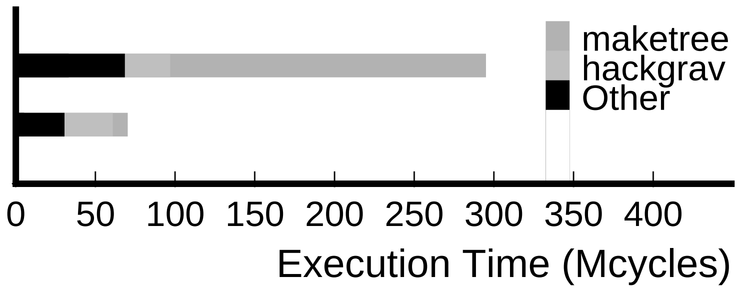


Figure 6.10: Breakdown of runtime in Barnes-Hut into major phases of computation. "maketree" is the parallel tree build phase, "hackgrav" is the force computation phase, and "Other" is all other computation. The top bar represents a 32-node DSSMP with an SSMP node size of 4, the bottom bar represents a 32-node Alewife machine.

interacting each body with the summary of a progressively larger number of bodies as interaction distance increases. A global octree data structure[7] enables the computation by hierarchically partitioning space and summarizing all bodies inside each partition using center-of-mass information (for a detailed discussion of the Barnes-Hut algorithm, see [60]).

To provide insight into what part of the Barnes-Hut workload is responsible for the high locking overhead in Figure 6.8, Figure 6.10 shows a breakdown of execution time into three components: the *maketree* routine builds a new octree data structure at each iteration, the *hackgrav* routine computes the force interactions by traversing the octree for each body, and "Other" represents all other work. The top and bottom bars in Figure 6.10 correspond to the 4-processor and 32-processor SSMP node sizes in Figure 6.8, respectively. This data clearly shows that the obstacle for DSSMPs is the *maketree* routine, which runs over 20 times slower on the DSSMP[8].

The poor performance in *maketree* is due to lock overhead. In *maketree*, locks enforce mutual exclusion for the simultaneous updates performed on the octree data structure. Extremely large lock overheads occur because of an effect that we call *critical section dilation*. For each locking operation, a processor obtains a lock, writes a value in the octree structure, and then relinquishes the lock. On a hardware DSM, these operations complete with very low overhead. However, on a DSSMP, a TLB fault or a page fault (or both) can be suffered on the updated location. Moreover, a release operation is required before the lock can be relinquished to make the updated value visible on all other

[7]An octree is a tree in which each node has a degree of 8.

[8]It is interesting to note that the *hackgrav* routine actually performs better on the DSSMP than on the all-hardware system. This is due to the ability of the software page cache in DSSMPs to capture the large working set in *hackgrav*, which reads large portions of the octree structure. The page cache converts misses in the hardware cache, which has insufficient capacity, from remote misses to local misses.

processors. Depending on sharing patterns, the release can initiate software coherence. These sources of software overhead combine to increase the cost of the locking operation. More importantly, they dilate the length of the critical section, or the time for which the lock is held by the processor. This tends to increase contention for lock resources, and when critical section length becomes large enough, processors can spend a significant amount of time serialized on locks.

A necessary condition for critical section dilation is poor data locality. If data accessed within critical sections are not shared across SSMPs, then MGS' Single-Writer mechanism will eliminate software-related overheads on the data and export hardware performance to critical section code. Critical section dilation is a problem in *maketree* because of poor locality on two data structures: the octree node allocation counter, and the octree nodes themselves.

Allocation of octree nodes occurs through a centralized counter that points to the head of a freelist of octree nodes. Processors allocate nodes off the head of the freelist by atomically incrementing the allocation counter. Because processors allocate nodes randomly and because of the high frequency of node allocation operations, the allocation counter becomes a hotspot. Once a node has been allocated, a processor inserts the node into the octree by atomically updating a child pointer in an existing octree node. Since processors tend to build entire subtrees of the octree, good data locality is expected for the node insertion operations; however, communication between SSMPs occurs nevertheless due to page-level false sharing. False sharing occurs because octree nodes are randomly allocated off a single freelist. Therefore, processors from separate SSMPs often receive octree nodes that physically reside on the same page. Updates to such distinct but contiguous nodes by different SSMPs causes page-level coherence.

In addition to lock overhead, Barnes-Hut also exhibits significant barrier and MGS overhead. The barrier overhead arises from load imbalance due to both algorithmic and MGS effects. Algorithmically, load imbalance occurs in Barnes-Hut because the amount of work associated with each body highly depends on the distribution of the bodies in space. Although Barnes-Hut attempts to dynamically load balance work (see [60] for details), the technique is not perfect thus accounting for some of the barrier overhead. Load imbalance can also arise due to MGS overhead. Handler occupancy for servicing a particular shared memory transaction occurs on the processor that is the home for that page. Therefore, processors that serve as the home for "hot" pages will carry a disproportionate fraction of the software shared memory processing load thus contributing to load imbalance. Finally, the source of MGS overhead in Barnes-Hut comes mostly from the write sharing patterns in the *maketree* routine.

Pathologic Category

TSP and Unstructured, presented in Figures 6.11 and 6.12, respectively, belong to the "Pathologic" category. Applications in the pathologic category have a similar performance profile as compared with applications in the challenging category in that they have both a large multigrain potential and a large breakup penalty. A key difference,

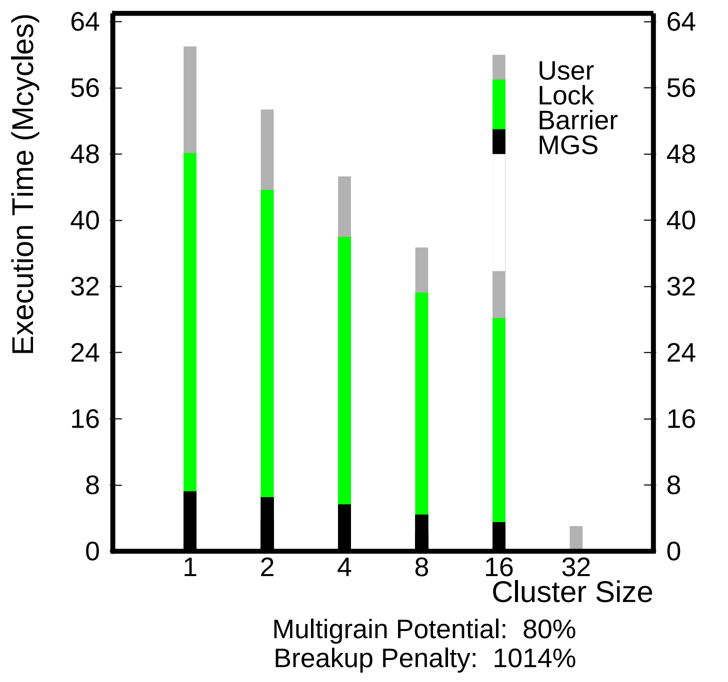


Figure 6.11: Results for TSP.

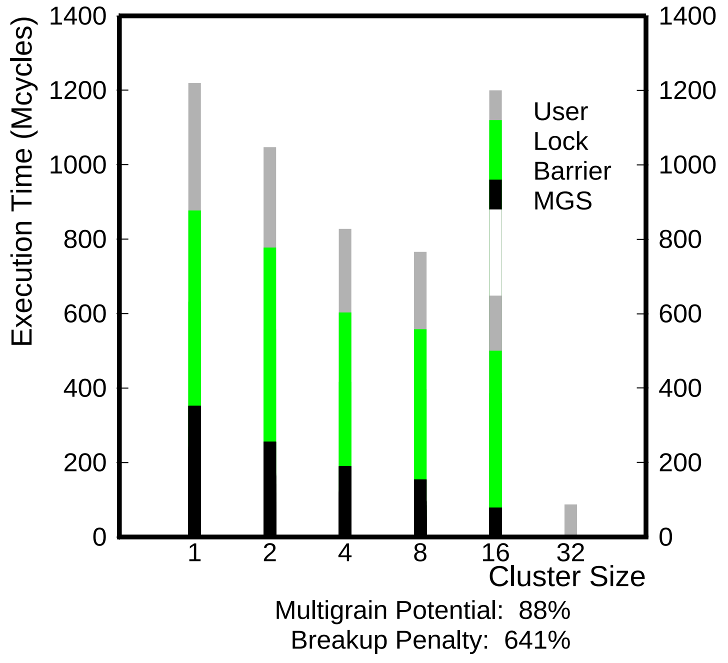


Figure 6.12: Results for Unstructured.

however, is that the breakup penalty in TSP and Unstructured is so large that the DSSMPs do not achieve any effective speedup on these applications, even as SSMP node size is increased. Table 6.6 shows the speedups for TSP and Unstructured are all below 2, with TSP exhibiting slowdown in the worse case.

Figure 6.11 shows that TSP suffers from extremely high lock overhead. The source of lock overhead in TSP is a centralized work pool data structure that dynamically distributes work across the machine. Each processor adds partially evaluated tours to the work pool and removes them when it runs out of work. To minimize the overhead associated with the work pool, only those partial tours that represent large amounts of work are added to the pool[9]. Still, the frequency of operations on the work pool is very high, and for the architectures that use software in supporting shared memory, the overhead associated with the work pool is large.

The work pool overhead shows up as lock overhead in Figure 6.11 because of critical section dilation on several locks used to provide mutually exclusive access to the work pool structure. The critical section dilation effect, which we saw earlier in the Barnes-Hut workload, significantly increases the cost of each locking operation. This leads to lock contention, which is particularly severe in TSP because there is only a single centralized work pool for the entire machine. Notice, however, that despite the contention on the centralized work pool data structure, the 32-node SSMP (all-hardware DSM) system manages to achieve decent performance nonetheless. This speaks volumes about the robustness of the DSM concept in the face of applications with poor locality characteristics.

The other application in the pathologic category is Unstructured, whose results appear in Figure 6.12. Unstructured is by far the most difficult application to achieve high performance on DSSMPs because of its highly irregular data access patterns. The application performs a computation on a static undirected graph which is read from an input file. Because of the graph's unstructured nature, runtime preprocessing techniques [55] are used to schedule computation associated with the graph onto processors. After preprocessing, much of the execution time is spent in *edge loops*, or loops that perform computations associated with the edges in the unstructured graph. Each iteration of an edge loop reads values from the two graph nodes connected by the edge, computes a result, and updates the result into the two graph nodes. Locking in the edge loops is used to provide mutually exclusive access to those graph nodes which are accessed by multiple processors (*i.e.* graph nodes with multiple edges assigned to different processors by the runtime preprocessing phase).

Unstructured is similar to the Water workload. Both are graph problems that perform computations on the edges of the graph. The key difference is in the structure of the graph. In Water, the graph is regular and includes all N^2 possible edges between graph

[9] The metric used to determine the amount of work represented by a partial tour is the number of cities already visited. If this number is small, the algorithm decides that the partial tour represents lots of work since there are many cities left to be visited. This metric is only an approximation since pruning can significantly reduce the amount of work associated with a partial tour.

nodes. In Unstructured, the graph is highly irregular, and is specified outside of the application.

The poor performance of Unstructured on DSSMPs is attributable to all three overheads reported in Figure 6.12—lock, barrier and MGS. The lock overhead component arises because the locks in the edge loops suffer from the now familiar critical section dilation effect. Irregularity in the input graph means that graph nodes are updated in a fairly random fashion as a result of the computation in edge loops. Random access of the graph nodes leads to poor data locality; therefore, a significant portion of the atomic updates to graph nodes invoke software coherence overhead thus resulting in critical section dilation. Barrier overhead is due to an imbalance in the schedule of edge computations inside the edge loops. While the runtime preprocessor tries to minimize load imbalance, it also tries to maximize data locality which is often a conflicting requirement. Finally, the MGS overhead component is due to poor data locality, most significantly in the edge loops that leads to critical section dilation.

6.4 Application Transformations

In this section, we further study the applications in the challenging (Water and Barnes-Hut) and pathologic (TSP and Unstructured) categories described in Section 6.3. The intent is to examine the bottlenecks that prevent higher performance in these difficult applications, and to investigate transformations that relieve these bottlenecks by leveraging fine-grain cache-coherent shared memory provided within SSMP nodes as often as possible. Once identified, the transformations are applied to the applications, and experimental results are presented to quantify the impact of the transformations. For consistency, the performance framework introduced in Section 6.2 is used to analyze DSSMP behavior on the transformed applications.

The applications study undertaken in this section has two primary goals. The first goal is to quantify the potential performance achievable on DSSMPs when details of the underlying implementation of shared memory are exposed to the application level, particularly SSMP node boundaries that delineate hardware- and software-supported shared memory. A crucial question related to this goal is whether such favorable conditions can make DSSMPs more competitive with MPPs. The second goal is to evaluate how plausible the proposed transformations are for programmers or preferably compilers to implement. We recognize that a transformation that provides good performance is meaningless if it takes an expert programmer several weeks to implement. We gauge the plausibility of transformations by looking for similar transformations that are within the capability of existing state-of-the-art compilers.

Section 6.4.1 discusses the transformations for the four applications, and Section 6.4.2 presents experimental results of the transformations. In Section 6.4.3, we discuss the plausibility of supporting the transformations in a compiler.

Application	Bottleneck	Transformation
Water-Kernel	Data Locality	Tiling Tile Scheduling
Barnes-Hut	Node Allocation Hotspotting False Sharing on Nodes Other Critical Section Dilation	Concurrent Allocation Distribute Freelist Add Releases
TSP	Contention on Work Pool	Distribute Work Pool
Unstructured-Kernel	Data Locality	Runtime Tile Analysis Runtime Tile Scheduling Runtime Load Balancing

Table 6.7: Summary of performance bottlenecks and transformations.

6.4.1 Transformation Descriptions

Table 6.7 lists the four applications from the challenging and pathologic categories: Water, Barnes-Hut, TSP, and Unstructured. For each application, bottlenecks that prevent higher performance are specified along with the transformation(s) that relieve the bottlenecks.

In the case of Water and Unstructured, we only study a kernel from the original applications. These kernels contain the portions of the application that suffer the performance bottlenecks observed in Section 6.3.2. Water-Kernel executes the force computation loop presented in Figure 6.9 once. Unstructured-Kernel executes a single edge loop. The baseline performance of these kernels on Alewife (with software address translation) can be found in Table 6.5.

The rest of this section discusses the transformations in detail.

Water

We address the data locality problems in the Water workload by first applying loop tiling to the force computation loop in Figure 6.9. In our loop tiling transformation, the global molecule array is partitioned into tiles, where each tile consists of $tile\_size$ contiguous molecules. Then, the original doubly nested loop nest is transformed into four loop nests. The two outermost loops iterate through all possible $(tile\_size)^2$ pairings of tiles. Given a pair of tiles, the two innermost loops compute the $(\frac{N}{tile\_size})^2$ possible pairwise interactions between two molecules, one from each tile. All processors in an SSMP execute the same two outermost loop nests; therefore, each SSMP works on a single pair of tiles. The loop body in the transformed loop nest remains unchanged from Figure 6.9.

Loop tiling improves data locality because all the interactions between a small group of molecules (*i.e.* the molecules in two tiles) are computed before moving on to interactions involving other molecules. To ensure that such improved data locality leads to less inter-SSMP coherence overhead, it is also necessary to address write sharing conflicts.

Write sharing conflicts are eliminated if each SSMP has exclusive ownership of the two tiles accessed during an iteration of the two outermost loops. For this to happen, we must first choose the *tile_size* parameter so that the number of tiles, $\frac{N}{tile\_size}$, equals twice the number of SSMP nodes. This makes it possible for every SSMP node to exclusively own two tiles. Then, we perform a tile scheduling transformation that orders the iterations of the two outermost loops in a staggered fashion across different SSMPs. The staggering pattern ensures that all the tiles are accessed in a conflict-free manner.

Barnes-Hut

Three transformations are proposed for the Barnes-Hut workload in Table 6.7. The first two transformations address poor data locality that lead to critical section dilation in the *maketree* routine. First, we relieve the hotspotting problem associated with the node allocation counter by removing the centralized counter and allowing concurrent allocation off the octree node freelist. In this transformation, there is still a single physical freelist, but concurrent allocation is made possible by statically allocating freelist entries to processors in an interleaved fashion. Next, we relieve the false sharing problems on octree nodes by physically distributing the centralized freelist such that each processor has its own local freelist. This guarantees that octree nodes contiguous in memory are allocated to the same processor thus eliminating false sharing between SSMPs.

The last transformation for Barnes-Hut in Table 6.7 relieves critical section dilation problems in computation outside of the *maketree* routine. These instances of critical section dilation are different from the ones that occur in *maketree* because they are not a result of poor data locality. Instead, they occur because a large number of pages are updated right before a critical section is entered. Therefore, when the release operation associated with the critical section is issued, it initiates coherence on not only the few pages modified inside the critical section, but also on all the extra pages previously touched. Our transformation adds a release before entering the critical section so that the overhead for processing the extra pages is incurred outside of the critical section.

TSP

The contention problems on the work pool data structure in the TSP workload are addressed by replacing the centralized work pool with a physically distributed work pool. The distributed work pool places a local work pool structure on each SSMP that is identical to the original centralized work pool. Processors add and remove partially evaluated tours to and from their local work pools just as before, except all contention on any single local work pool occurs only between processors in the same SSMP; therefore, software shared memory is avoided entirely for local work pools.

A centralized work pool structure is still used to distribute work globally; however, contention on this data structure is low because most of the work pool operations are offloaded to the local work pools. Only partially evaluated tours representing very large amounts of work are added to the global work pool, and work is removed from the global work pool only when work from a local work pool has been completely exhausted.

Unstructured

The tiling transformation for Unstructured groups graph nodes into tiles of size $tile\_size$. As in Water, $tile\_size$ is chosen such that the number of tiles, $\frac{N}{tile\_size}$, is equal to twice the number of SSMP nodes. At runtime on each SSMP, the graph edges assigned to processors on the same SSMP are sorted into a list of bins where each bin represents a unique pairing of tiles. In a fully connected graph such as the one in the Water workload, the size of each bin is $(tile\_size)^2$. In Unstructured, each bin contains a different number of edges[10] depending on the structure of the graph. The bins defined by our runtime tile analysis are used to drive the order in which edges are computed in the edge loops. Specifically, processors compute all the edges in a bin before computing edges in the next bin. Furthermore, barriers are used to sequence processors through their list of bins in lockstep fashion.

Next, runtime analysis is used to schedule tile interactions by computing an order for the lists of bins on each SSMP. The ordering must ensure that tile conflicts do not occur between bins at the same list position across all SSMPs. A greedy algorithm for scheduling bins is used which considers the largest unscheduled bin from each list first. A bin is scheduled if the two tiles it interacts is not needed by any currently scheduled bin at the same list position. If a bin cannot be scheduled, then the next largest unscheduled bin is considered, and so on. If no bin can be scheduled after considering all remaining bins, a bubble is placed in the schedule.

Finally, an attempt is made to load balance the results of the runtime tile schedule by spreading the edges in each bin evenly across all the processors in a single SSMP. While near-perfect load balance between the processors within an SSMP can be achieved, load imbalance across SSMPs due to varying bin sizes is not addressed. Our primary goal between SSMPs is to maximize data locality, if necessary at the expense of load balance.

6.4.2 Transformation Results

This section presents the performance of the four applications, Water-Kernel, Barnes-Hut, TSP, and Unstructured, after the transformations described in Section 6.4.1 were applied. The goal is to quantify the impact of each transformation on application performance. Whenever multiple transformations are involved on a single application, we apply the transformations one at a time to assess the importance of each individual transformation. As in Section 6.3.2, inter-SSMP communication latency is set at 1000 cycles (50 μsec), and page size is set at 1K-bytes. Section 6.5 later examines the impact of varying communication latency and page size on performance.

Overall Results

The overall results of the application transformations are summarized in Table 6.8. Table 6.8 presents the data in a fashion similar to Table 6.6, giving the multigrain potential,

[10] In fact, many of the bins are empty.

Transformation	MP	BP	SVM	S1	S2	S4	S8	S16
Water-Kernel								
Original	125	120	58.47	5.3	6.3	7.4	9.9	11.9
Tiling & Tile Scheduling	58	24	56.95	13.7	16.7	19.6	21.3	21.6
Barnes-Hut								
Original	61	193	72.77	2.8	3.0	3.3	3.6	4.6
Concurrent Allocation	148	48	65.47	4.1	4.8	5.8	7.1	10.1
Distribute Freelist	83	50	60.40	5.9	6.8	7.6	8.1	10.8
Add Releases	81	35	60.40	6.6	7.9	8.9	9.5	11.9
TSP								
Original	80	1014	3.04	0.9	1.0	1.2	1.5	1.6
Distribute Work Pool	282	66	6.80	2.8	5.2	9.1	11.7	10.6
Unstructured-Kernel								
Original	100	537	13.44	1.2	1.5	1.9	2.0	2.4
R.T. Tile Analysis & Scheduling	402	25	28.30	1.1	2.4	2.5	3.9	5.8
R.T. Load Balancing	812	38	13.30	1.2	3.7	4.8	7.2	11.1

Table 6.8: Summary of application transformations performance on DSSMPs. The "MP" column reports Multigrain Potential, the "BP" column reports Breakup Penalty, and the "SVM" column reports running time on a 32-processor Alewife machine with SVM overhead in millions of cycles. The last five columns report speedups on SSMP nodes of size 1–16, in powers of two.

breakup penalty, and speedups achieved on different SSMP node sizes. In addition, the "SVM-Par" column reports the parallel running time on a 32-processor Alewife machine with software address translation but without other MGS-related overhead, similar to Table 6.5. The SVM-Par column shows the impact of each transformation on an all-hardware DSM.

For each application in Table 6.8, the row labeled "Original" presents the performance results before the transformations are applied. For Barnes-Hut and TSP, these results are identical to the ones shown in Table 6.6. For Water-Kernel and Unstructured-Kernel, the "Original" numbers are different since they involve the kernel versions of the applications. Each subsequent row following "Original" shows the incremental effect of applying one of the transformations from Table 6.7. We note that all speedups are computed by using the numbers under the "SVM-Seq" column in Table 6.5; therefore, the speedups represent speedup over the original sequential application.

The overall results presented in Table 6.8 provide two positive conclusions. First, the application transformations are very effective in reducing the breakup penalty, which is the key feature that makes these applications difficult. After the transformations are applied, Water-Kernel, Barnes-Hut and Unstructured-Kernel have breakup penalties below 40%; TSP has a slightly higher breakup penalty at 66%. The reduction in breakup penalties allow the speedups observed on DSSMPs to approach the speedups observed on Alewife, as reported for these applications in Table 6.5. We conclude that properly

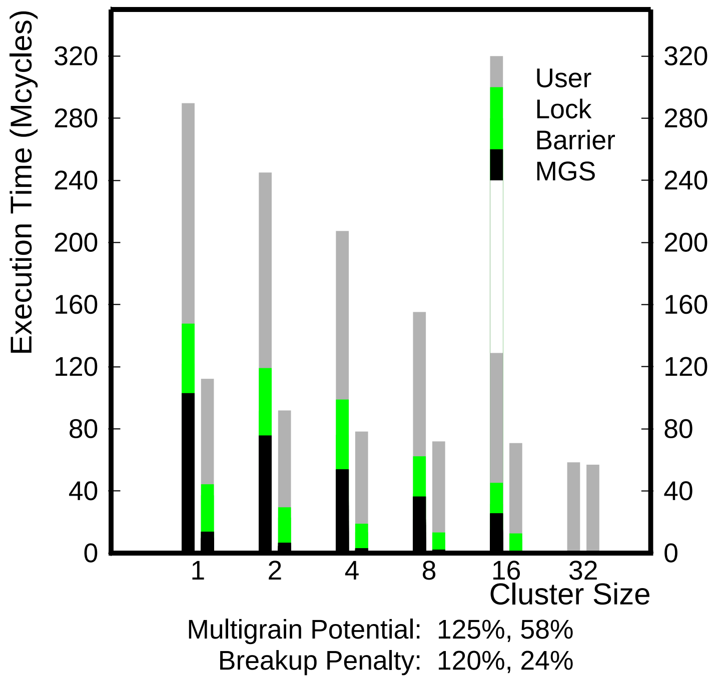


Figure 6.13: Transformation results for Water.

structured applications can achieve comparable absolute performance on DSSMPs as compared with all-hardware shared memory systems.

Although the breakup penalties have been reduced, Table 6.8 also shows that the multigrain potentials are still high, even after the transformations have been applied. In fact, in all the applications except for Water, the multigrain potential increases, and in Water, it is still rather significant at 58%. We conclude that even for optimized programs, having some fine-grain hardware-supported shared memory within SSMP nodes is beneficial. Our data show that transformations can leverage this clustered hardware support to achieve higher performance than on an all-software DSM.

Detailed Results

Figures 6.13 through 6.16 present detailed results for the application transformations using our performance framework. The presentation format is identical to the one used in Section 6.3.2, except that we plot the results of the original application with the transformed versions of the application, side by side.

Figure 6.13 shows the detailed results for the Water-Kernel workload, before (left set of bars) and after (right set of bars) the tiling and tile scheduling transformations are performed. The effectiveness of these transformations to improve data locality is demonstrated by the reduction in the MGS overhead component. The transformations are more effective on larger SSMP nodes, accounting for the multigrain potential, because

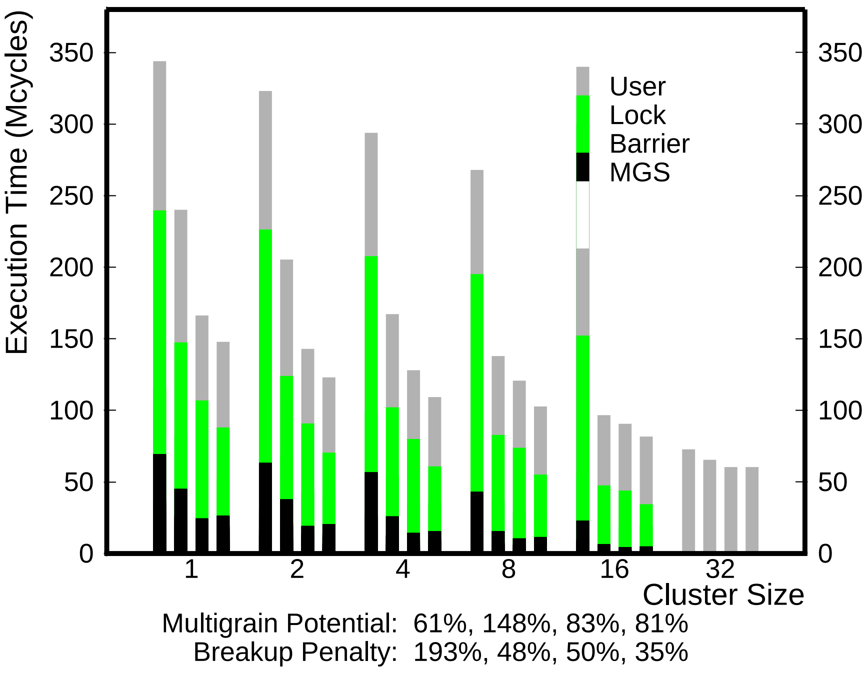


Figure 6.14: Transformation results for Barnes-Hut.

tile size grows with SSMP node size. Larger SSMP nodes, and thus larger tiles, mean that more computation within SSMP nodes is possible before inter-SSMP communication must occur to communicate tiles. Such an increase in computation-to-communication ratio gives a performance advantage to larger SSMP nodes.

Figure 6.14 shows the results for transformations on Barnes-Hut. The four bars at each SSMP node size report the performance observed on the original application and the three transformations. It is interesting to note the relative efficacy of the three transformations. The first transformation to remove contention on the centralized octree node allocation counter (2nd set of bars) significantly improves performance. The second transformation to eliminate false sharing on octree nodes (3rd set of bars) also significantly improves performance, though its effects are most pronounced at smaller SSMP node sizes. Finally, the transformation for critical section dilation outside of the *maketree* routine (rightmost set of bars) produces the least gain in performance.

TSP, shown in Figure 6.15, displays the greatest gains in performance as a result of its transformations, largely because it was the application with the worst performance to begin with. Figure 6.15 clearly shows that the enormous overheads associated with the centralized work pool can be eliminated if a more scalable data structure is used instead. There is some unexpected behavior in Figure 6.15 at SSMP node sizes 16 and 32—performance decreases with increasing SSMP node size, and the transformation actually worsens performance on an all-hardware DSM. This anomaly is an artifact of the distributed work pool implementation. When there is only a single SSMP node, the

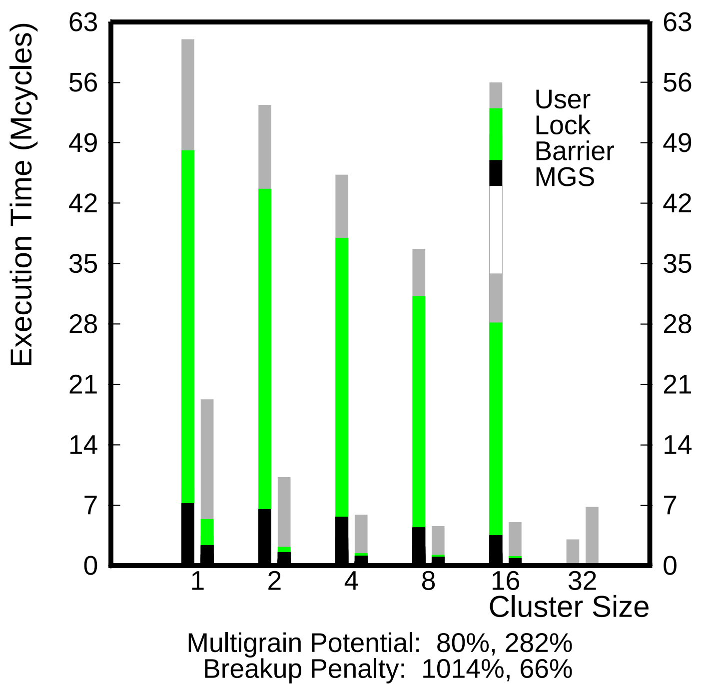


Figure 6.15: Transformation results for TSP.

global work pool serves no purpose (since it is meant to distribute work between SSMP nodes). However, it artificially decreases parallelism because work is not removed off the global work pool until all work from the local work pool has been consumed.

Finally, Figure 6.16 shows transformation results for the Unstructured-Kernel workload. Three bars at each SSMP node size show the performance of the original application and the two transformations. The first transformation (2nd set of bars) improves data locality by performing a runtime tiling transformation. Performance improves for most of the cluster configurations. However, the improvement of data locality in this transformation comes at the expense of load imbalance. Load imbalance becomes more severe at smaller SSMP node sizes because the smaller tiles used for smaller SSMP nodes lead to a greater variability in the number of interactions between two tiles. Therefore, the performance at an SSMP node size of one actually gets worse as a result of the transformation. Another consequence of load imbalance is performance degrades significantly for the all-hardware case.

The second transformation (3rd set of bars) partially addresses the load imbalance problem created by the tiling transformation. In particular, good performance is restored to the all-hardware case, and significantly better performance is observed on all SSMP node sizes except for the SSMP node size 1. Our runtime load balancing transformation only attempts to load balance between processors in the same SSMP, since we didn't want to sacrifice data locality between SSMPs. When there is only a single processor per SSMP node, the load balancing transformation is useless.

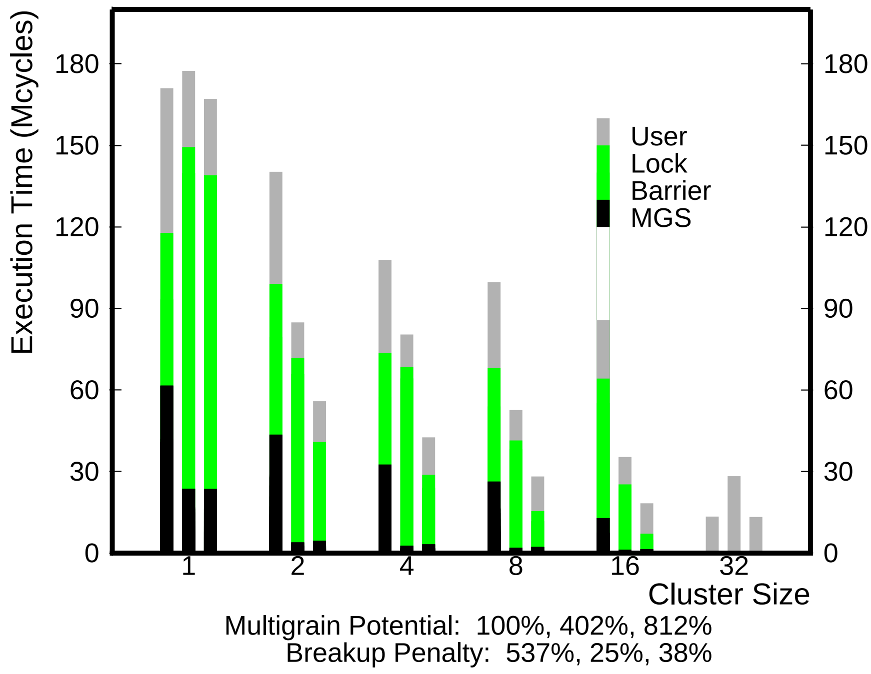


Figure 6.16: Transformation results for Unstructured.

6.4.3 Discussion

Section 6.4.2 reports impressive performance gains for the application transformations described in Section 6.4.1. In this Section, we discuss the sophistication of these transformations, and consider the feasibility for compilers or programmers targeting DSSMPs to implement them.

In the Water workload, we implemented a loop tiling and tile scheduling transformation. Similar transformations have been proposed in the compiler literature. In particular, loop tiling has been studied in [69] as a technique for improving data locality on parallel codes. Other work [4, 70] has looked at loop tiling as well, which is sometimes referred to as "strip-mine and interchange" and "unroll and jam" transformations. The technique is mature enough that many parallelizing compilers already perform loop tiling automatically. We believe the tiling transformations performed for Water in this thesis can be automated using similar techniques already published.

The Barnes-Hut workload received three transformations. The first transformation addresses the hotspotting problem on the centralized node allocation counter. To our knowledge, there is no automatic technique for addressing such hotspotting problems in existing compilers. However, it is arguable that such an unscalable implementation for a data structure that is frequently accessed constitutes poor programming practice. In fact, the authors of the original code for Barnes-Hut fixed the problem in a fashion almost identical to our transformation in a re-release of their application suite [71]. Furthermore, the transformation we apply is trivial, and can be easily implemented once the problem is identified. We anticipate that such contention problems on shared memory objects can be addressed by the programmer aided by performance monitoring tools.

The second transformation for Barnes-Hut addresses the false sharing problems on the octree nodes. False sharing on shared memory systems has been studied quite extensively. Most notably, the work in [33] proposes analysis techniques for a compiler that automatically detects and eliminates several types of false sharing access patterns. The transformation we implement for Barnes-Hut is very close to a transformation that is handled by their analysis, known as "group and transpose."

The last transformation for Barnes-Hut deals with critical section dilation by adding extra releases before a critical section is entered. No existing compilers address this kind of transformation. One possible approach for this transformation is to conservatively apply it everywhere. The only possible negative side effect of such a conservative approach is the added overhead of the release operation in those cases where the transformation is not necessary. However, the overhead is fairly small since the overhead of maintaining coherence is suffered regardless of whether the extra release is inserted or not. The only added overhead is an extra kernel crossing into the MGS routine that handles releases. In the case of Barnes-Hut, this transformation may be omitted since it did not make a significant impact on performance.

Our experience with TSP is similar to the hotspotting problem in Barnes-Hut. Once again, an unscalable implementation of a frequently accessed data structure is the problem. While it is unlikely that the transformation we implemented could ever be performed by a compiler, we believe this is another example of poor programming practice that can be addressed by the programmer given proper feedback from performance tools.

Finally, we implemented runtime tile analysis, runtime tiling, and runtime load balancing for the Unstructured-Kernel workload. These transformations are highly sophisticated and require deep understanding of the application. Even with significant knowledge of Unstructured, it took the author of this thesis several weeks to diagnose the performance bottleneck and implement the transformation to relieve the bottleneck. For Unstructured, high performance on DSSMPs can only be attained by significant programming effort.

6.5 Sensitivity Study

Sections 6.3 and 6.4 report experimental experience with applications on our prototype of the MGS architecture assuming a fixed inter-SSMP communication latency and page size. In this section, we examine the sensitivity of MGS performance when these system parameters are varied. Section 6.5.1 discusses the impact of varying inter-SSMP communication latency, and Section 6.5.2 discusses the impact of varying page size.

6.5.1 Inter-SSMP Latency

The measurements obtained in Sections 6.3 and 6.4 assumed a fixed inter-SSMP communication latency of approximately 1000 cycles, which on a 20MHz Alewife machine translates to 50μsec. In this section, we investigate the sensitivity of application performance when inter-SSMP communication latency (one-way) is varied between 0 and 20,000 Alewife cycles, or between 0 μsec and 1 $m$sec. Our study considers the Jacobi Water, and Water-Kernel (with tiling) applications. For all our experiments, we use the same workload parameters reported in Table 6.4.

Figure 6.17 shows the impact of varying inter-SSMP communication latency for the Jacobi application. In Figure 6.17, execution time (in millions of cycles) is plotted against inter-SSMP communication latency (in thousands of cycles) for three different SSMP node sizes–1, 4, and 16 processors. The range of inter-SSMP latencies considered, 0 through 20,000 cycles, represents the *additional* latency on top of the baseline latency using Alewife communications interfaces. Note that the added latency we plot for each point along the X-axis is approximate because we use the average latency reported by MGS across all inter-SSMP messages sent during the lifetime of the application. Slight variations can occur across individual messages due to system effects[11]. The sensitivity of application performance to communication latency can be measured by observing the slope of the curves in Figure 6.17.

We see from Figure 6.17 that Jacobi is fairly insensitive to variations in inter-SSMP communication latency. The slope of the curves are very small indicating very slight impact on application runtime. The execution time changes by only 7% for both the 4 and 16-processor SSMP node sizes, and by 14% for the 1-processor SSMP node size across the entire range of 1 $m$sec of latency. Jacobi performs very little communication relative to the amount of computation in the application; therefore, changes in the communication latency has very little impact on overall application performance.

The sensitivity to communication latency is much higher in the Water application since Water communicates much more frequently than Jacobi. Furthermore, the sensitivity depends drastically on the SSMP node size; DSSMPs built with smaller SSMP nodes are much more sensitive than DSSMPs built with larger SSMP nodes due to the reduction in inter-SSMP communication that occurs when SSMP node size is increased. While this effect can be observed on Jacobi, it is much more pronounced for Water because the larger communication volume in Water means there is a greater potential for communication reduction through clustering. Figure 6.18 shows that execution time on Water increases by 294%, 202%, and 95% across the 1 $m$sec range of latency for SSMP node sizes of 1, 4, and 16 processors, respectively. Through linear interpolation, we can approximate the communication latency required to maintain a certain level of performance at each SSMP node size. For the performance impact to be within 10% of the

[11] For instance, the timer facility used to simulate delayed messages (see Section 5.3.2) cannot generate an interrupt if interrupts are off the moment it expires. In this case, the interrupt will be deferred until interrupts are turned back on. Such system effects can increase the variance in the latency observed across multiple delayed messages.

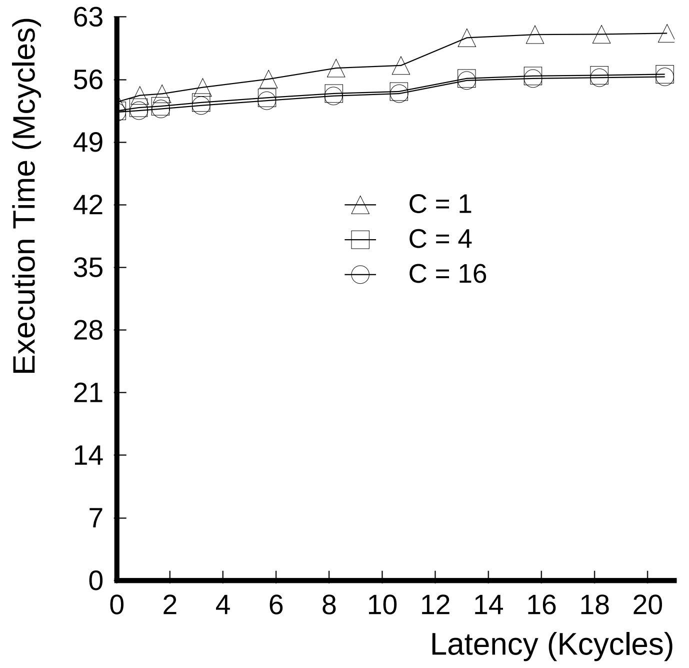


Figure 6.17: Latency sensitivity results for Jacobi.

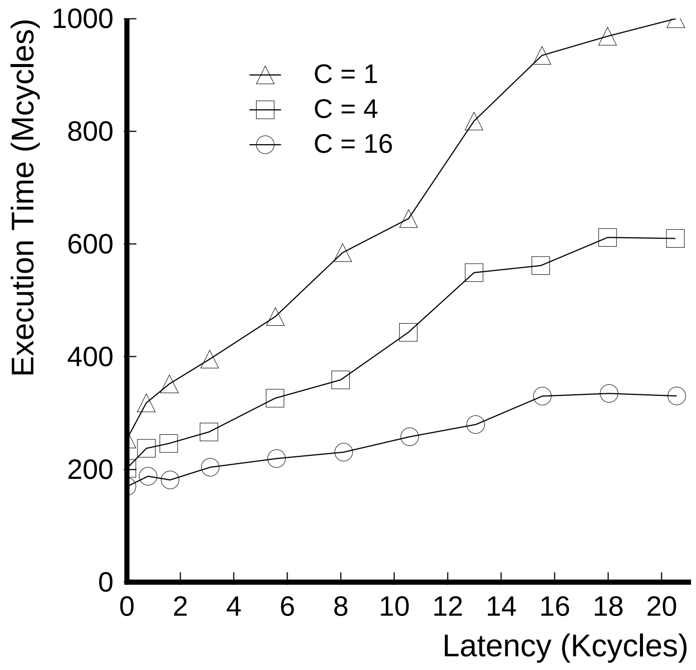


Figure 6.18: Latency sensitivity results for Water.

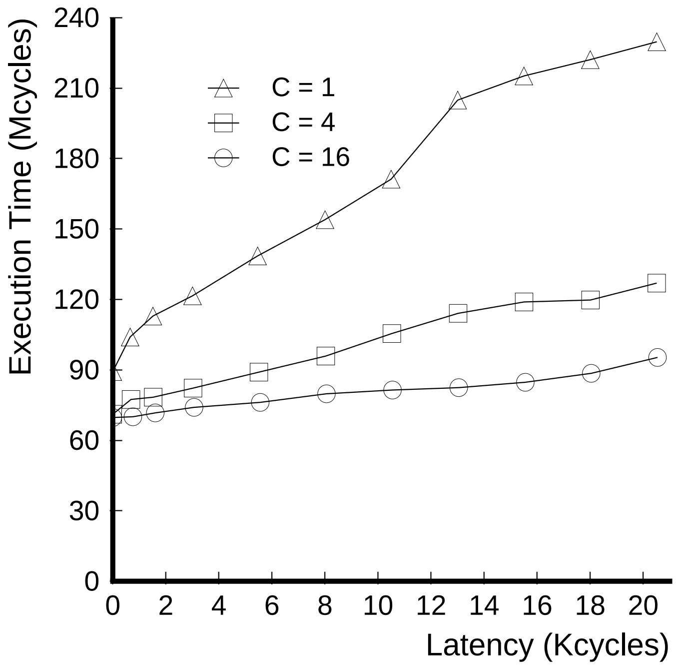


Figure 6.19: Latency sensitivity results for Water-Kernel with tiling.

performance experienced using Alewife communications interfaces, the inter-SSMP communication latency must be no greater than 34.8, 50.8, and 108.8 μsec (695, 1015, and 2175 cycles) for SSMP node sizes of 1, 4, and 16 processors, respectively. While these are aggressive numbers, they are achievable using existing communications interfaces.

As we saw in Section 6.4, locality-enhancing transformations can significantly increase DSSMP performance. Since these transformations reduce the use of software page-based shared memory, the transformed applications should be less sensitive to inter-SSMP communication latency as well. Figure 6.19 illustrates the sensitivity of the Water-Kernel code to communication latency after tiling has been applied, as described in Section 6.4.1. Execution time for Water-Kernel with tiling changes by 157%, 79%, and 37% across the 1 $m$sec range of latency for SSMP node sizes of 1, 4, and 16 processors, respectively. As expected, the sensitivity to communication latency in Water-Kernel with tiling is less than what was observed for the original Water application in Figure 6.18. However, sensitivity is still significant, especially for small SSMP node sizes in which the transformation is less able to reduce inter-SSMP communication.

6.5.2 Page Size

The measurements obtained in Sections 6.3 and 6.4 assumed a fixed page size of 1K-bytes. In this section, we examine the effect of varying page size on MGS performance. These experiments are made possible by the fact that MGS supports address translation

System	Cache Line	Page	Ratio
MGS	16	1K	64
Ultra SPARC (I & II)	64	8K, 64K, 512K or 4M	128–64K
MIPs R10000	64 or 128	4K–16M in powers of 4	32–256K
Alpha 21164	32 or 64	8K	128–256

Table 6.9: Relative grain on MGS and other systems.

in software. We begin by discussing the issues involved with selecting a page size, and then present results for varying page size on the Water-Kernel code.

Selecting a Page Size

Our goal for the detailed results in Sections 6.3 and 6.4 was to make them as representative as possible of DSSMP performance regardless of the specific SSMP hardware platform used. The choice of the page size is an important factor in meeting this goal since DSSMP performance, in general, changes depending on the specific page size selected, as the results later in this section will show. 1K-bytes, the page size used for the detailed results presented earlier in this chapter, is a fairly small page size, particularly for modern machines. However, our selection of page size was not intended to match the absolute page sizes of modern machines; instead, we were concerned with matching the ratio of page size to cache-line size. This ratio is the number of intra-SSMP coherence units delivered for each inter-SSMP coherence unit. To a first order approximation, this ratio determines the number of page faults relative to the number of cache misses. Because Sparcle represents a processor that is 2 to 3 generations removed from the present, it has a small cache-line size compared to modern processors. To maintain a reasonable page size to cache-line size ratio, we selected a smaller page size.

Table 6.9 compares the cache-line and page sizes in our MGS prototype with those found in systems based on the Ultra SPARC, MIPS R10000, and Alpha 21164 processors, three modern processors. The table shows the cache-line and page sizes on all the systems. The cache-line sizes are those found in the second-level cache (except for MGS, since Sparcle only has one level of caching) which is the coherence unit that would be used between processors in a cache-coherent multiprocessor. The MIPs and Alpha processors support multiple second-level cache-line sizes, and the SPARC and MIPs processors support multiple page sizes, as indicated in the table. The last column in Table 6.9, labeled "Ratio," indicates the number of cache-lines in each page. In those instances where multiple cache-line and page sizes are supported, a range of ratios is given showing the minimum and maximum ratios.

As we will see later in this section, selecting the page size involves making a tradeoff between more severe false sharing effects at large page sizes, and amortization of software overhead over less data at small page sizes. In general, the false sharing effects are more severe in those applications that are susceptible to false sharing than the software overhead amortization effects, so it is desirable to have a smaller page size. Therefore, DSSMPs built on the platforms listed in Table 6.9 would use the smallest page sizes supported by each processor architecture, resulting in the minimum ratio values. Comparing all the minimum ratio values in Table 6.9, we see that the ratio between page size and cache-line size in MGS is within the range of what's possible on the MIPs platform, and only slightly smaller than what's possible on the SPARC and Alpha platforms.

Page Size Sensitivity Results

We examine the impact of selecting three different page sizes, 1K-bytes, 2K-bytes, and 4K-bytes, on MGS performance using the Water-Kernel code, both with and without tiling transformations. We use the workload parameters for Water-Kernel reported in Table 6.4.

Figure 6.20 shows the results for the original version of Water-Kernel (without tiling). We plot the execution time of Water-Kernel as SSMP node size is varied. At each SSMP node size, three bars are reported representing the three page sizes used (1K-bytes to 4K-bytes from left to right). We only report the 1K-byte execution time for the 32-processor SSMP node size configuration since in this DSSMP configuration, the system does not perform any paging.

Without the tiling transformation, Water-Kernel displays significant amounts of inter-SSMP communication. This communication volume is aggravated as page size is increased due to false sharing. The effect is relatively small going from a 1K-byte page size to a 2K-byte page size. On average, runtime increases by 15% across the SSMP node sizes between 1 and 16 processors. However, the impact is significantly more pronounced when going from a 2K-byte page size to a 4K-byte page size. In this case, execution time increases by 39% on average across the different SSMP node sizes.

While Figure 6.20 shows that increasing page size can negatively impact MGS performance due to an increase in false sharing, Figure 6.21 shows that increasing pagesize can improve MGS performance when the application exhibits good locality. Figure 6.21 shows the impact of varying page size in the Water-Kernel code after the tiling transformation has been applied, as described in Section 6.4.1. Tiling improves the data locality of the force interaction computation loop. Due to the improved data locality, the code does not exhibit false sharing. As a result, performance actually improves with increasing page size for all the data points (except for the 4K-byte data point at an SSMP node size of 1 processor in which performance degrades slightly). Performance improves because there are fewer page faults due to the fact that each page fault brings an increasing amount of data as page size is increased.

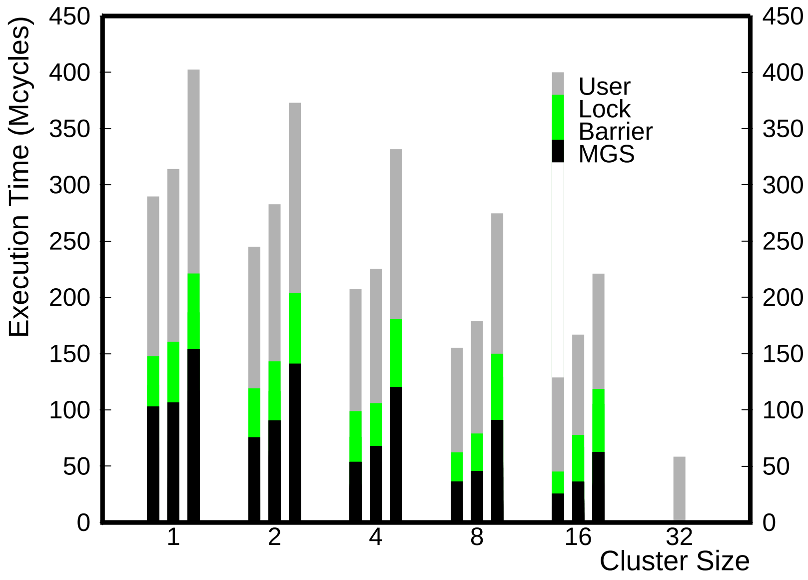


Figure 6.20: Page size sensitivity results for Water-Kernel. Total machine size is 32 processors. For each SSMP node size, the three bars show (from left to right) the performance attained when using a page size of 1K-bytes, 2K-bytes, and 4K-bytes, respectively.

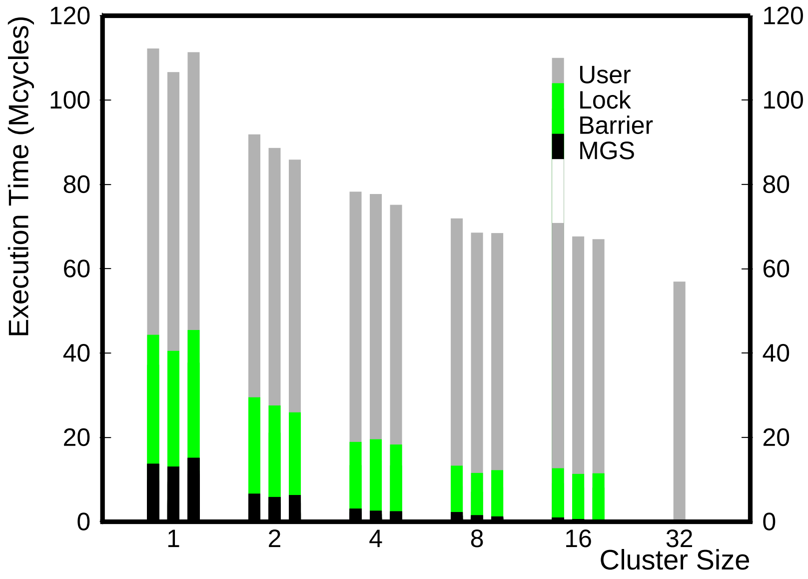


Figure 6.21: Page size sensitivity results for Water-Kernel with tiling. Total machine size is 32 processors. For each SSMP node size, the three bars show (from left to right) the performance attained when using a page size of 1K-bytes, 2K-bytes, and 4K-bytes, respectively.

Chapter 7

Analysis

System evaluation is a crucial component in the life cycle of a systems project. It is through evaluation that researchers characterize the overall behavior of a system, and validate the initial design goals that precipitated the system in the first place. For many systems projects, evaluation relies solely on experiments. Experiments can involve whole benchmarks that represent typical workloads, or the experiments can be carefully designed synthetic kernels that exercise specific aspects of the system in a controlled fashion. In either case, the product of the evaluation is experimental measurements which quantify system behavior. When experiments are properly performed, the measurements they yield provide the most objective evaluation possible of system behavior[1].

An alternative to experimental evaluation is analytic evaluation. Analytic evaluation is less direct than experimental evaluation. Instead of measuring system behavior, analytic evaluation relies on mathematical models to predict system behavior. Models are typically hypothesized and then validated against an existing system. The process of validation involves presenting both the model and the system with similar inputs and verifying that the output of the model matches the aspect of system behavior being modeled. Once validated, the model can be used to predict the system's behavior in the absence of the system.

While not as objective as experimental evaluation, analytic evaluation provides value in two ways. First, analytic approaches are more flexible than experimental approaches. Since analysis only involves the manipulation of mathematical expressions, it is feasible to vary system parameters. This is not possible in experimental approaches since they assume some kind of physical prototype (upon which to run the experiments) that is fixed[2]. Flexibility allows analytic approaches to study the reaction of system behavior to technological trends, or to study large systems that are beyond what is feasible to build in the lab. Second, analysis leads to insight that often eludes experimental evaluation. The hypothesis of a system model demands careful examination of all factors that contribute

[1] As the saying goes, "it's hard to argue with numbers."

[2] By physical prototype, we either mean a piece of experimental hardware, or a simulator of the target system running on a workstation. The inflexibility of experimental evaluation is less true in the case of simulators; however, mathematical models are still more flexible than simulators.

to overall system behavior. Validation provides feedback that either supports or refutes a particular hypothesis. By iterating through the hypothesis-validation cycle, all first-order effects that govern system behavior are eventually identified. This process of model formulation provides a deep understanding of the system, and can reveal key aspects of the system that would otherwise go undetected.

In this section, we develop and apply analytical techniques to further study the behavior of multigrain shared memory systems. In so doing, we hope to attain the benefits described above that analysis provides on top of experimental evaluation. In particular, this chapter makes the following contributions.

- **Analytical framework**. We develop a system model that predicts runtime of applications on multigrain systems. In developing this model, we identify several key attributes that govern performance on multigrain shared memory systems in particular, and release consistent shared memory systems in general.
- **Scalability study**. We use the model described above to study the scalability of multigrain shared memory systems. We investigate analytically the impact of scaling machine size. We are particularly interested in the relationship between SSMP node size and machine size, and the impact this relationship has on performance on very large machines.
- **Multigrain performance tools**. By developing analysis techniques, this chapter lays the groundwork for general multigrain performance tools. The techniques presented in this chapter can be used by programmers to evaluate software designs, or compilers to drive optimizations targeted for multigrain systems.

The rest of this chapter consists of two major parts. First, Section 7.1 describes the analytical framework. Second, Section 7.2 describes our scalability study.

7.1 Analytical Framework

This section describes the framework that enables analysis of multigrain shared memory behavior. We begin by looking at the problem of performance analysis in distributed shared memory systems. Section 7.1.1 briefly discusses why performance analysis for DSMs is difficult. Then, Section 7.1.2 argues that software DSMs supporting a release consistent shared memory model present a new opportunity for analysis which is not possible for other types of DSMs (*e.g.* hardware DSMs). The section identifies several key properties in software RC systems that make their analysis tractable. Finally, Section 7.1.3 presents a performance model that applies the discussion in Section 7.1.2 to predict application runtime on MGS.

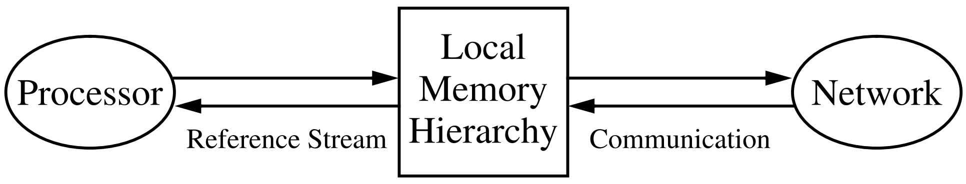


Figure 7.1: The local memory hierarchy in a DSM sits between the processor and the network.

7.1.1 Analyzing performance on DSMs

The crux of performance analysis for distributed shared memory lies in being able to accurately predict communication. This is important because of the difference in cost between a shared memory access that can be satisfied locally versus one that requires communication with a remote node. In MGS, the differential is particularly large (roughly 3 orders of magnitude as we saw in Section 6.1 of Chapter 6) because remotely satisfied accesses require software intervention whereas locally satisfied accesses leverage hardware.

Prediction of communication in DSMs involves the prediction of two aspects of system behavior. First, we must predict the shared memory reference stream emitted by each processor in the DSM. The shared memory reference stream is the sequence of shared memory requests that the processor emits to the memory system. Predicting this reference stream involves analyzing a shared memory application, identifying points in the code where shared memory references are performed, and for those references, determining the location referenced or the shared memory address. In general, this is a difficult task; further discussion will be provided in Sections 7.1.2 and 7.1.3.

Second, we must predict the behavior of the local memory hierarchy. As illustrated in Figure 7.1, the local memory hierarchy, which consists of possibly multiple levels of either hardware or software caches or both, sits between the local processor and the network. Each shared memory request emitted by the local processor is intercepted by the local memory hierarchy and is either satisfied by the local cache(s), or satisfied remotely via communication through the network with a remote node. Therefore, knowing the shared memory reference stream is not sufficient to predict communication; it is also necessary to know which references will miss in the cache(s) and thus cause communication.

In hardware cache-coherent shared memory systems, a shared memory reference can miss in a hardware cache due to one of four possible reasons corresponding to the four different types of cache misses: cold, capacity, associativity, and coherence misses[3]. Cold misses are the simplest and depend solely on the reference stream of the local processor. Capacity and associativity misses, however, result from the interaction of the locality properties in the local processor's reference stream with the size and organization of the local processor's hardware cache. And coherence misses, the most complex miss type to analyze, arise due to the interleaving of references in the local processor's reference

[3]We assume the reader is familiar with these terms. We refer the interested reader to [28] for a detailed explanation.

stream with conflicting references performed by other processors in the system.

Cache miss behavior, and thus communication behavior, in hardware DSMs is dictated by a complex interaction between different processor reference streams, and the architecture of the local memory hierarchy. Due to their complexity, these interactions are difficult to analyze. Consequently, even if we can effectively predict a local processor's reference stream on a hardware DSM, it is unlikely that we can predict the communication behavior.

7.1.2 Communication in Software DSMs

We believe there is a greater opportunity for communication analysis in software DSM systems. Fundamentally, the problem with analysis in hardware DSMs is that communication on each node has only an indirect relationship with the reference stream emitted by the local processor due to the intervention of an unpredictable system layer, as illustrated in Figure 7.1. In the following sections, we argue that there is a much more direct relationship between communication and processor reference streams in software DSMs. The key is that processors have more control over the contents of their local caches thus removing much of the unpredictability introduced by caching in hardware DSMs.

Cold, Capacity, and Associativity Misses

One of the reasons why analysis of communication in software DSMs is easier than in hardware DSMs is because two of the cache miss types described for hardware DSMs in Section 7.1.1, capacity misses and associativity misses, can be ignored without loss of analysis accuracy. Capacity misses are insignificant in software DSMs (though they can occur) because the amount of storage available for caching can be extremely large. Software DSMs perform caching in main memory, so it is feasible to make cache sizes on the order of 100s of megabytes or larger. For most applications, this is effectively an infinite cache. Hardware caches cannot be nearly as large because they are limited by chip area and clock speed design constraints. In addition to ignoring capacity misses, associativity misses can be ignored as well. Since virtual memory systems allow a virtual page to be placed in any physical page frame, the page cache in a software DSM is fully associative; therefore, software DSMs never suffer associativity misses. Most hardware caches are not fully associative and are susceptible to associativity misses, once again due to area and speed constraints.

We can further simplify communication analysis in software DSMs by ignoring cold misses. The number of cold misses suffered by a software DSM is exactly the number of unique data pages touched by the application. For most applications, this number is negligible compared to the total number of misses (or page faults) incurred by the application. Our analysis presented in Section 7.1.3 does handle cold misses, but for the applications studied in this thesis, cold misses do not impact the performance of the overall application. We should note that this assumption is not unique to software DSMs and can be applied to hardware DSMs as well.

By ignoring the effects of cold, capacity, and associativity misses, the analysis of cache miss behavior for software DSMs is greatly simplified. Once a page is placed in a local processor's page cache, we are guaranteed that it will stay there until it is invalidated by a remote processor. Therefore, to analyze communication on software DSMs, it is only necessary to track coherence misses.

Coherence Misses

Coherence misses are the most difficult of the four cache miss types to analyze. To account for coherence misses, the analysis must perform two tasks. First, the analysis must identify shared memory accesses performed on different processors that conflict. Two accesses conflict if the locations accessed fall on the same cache block (*i.e.* page for software DSMs), and at least one of the accesses performs a write. Second, the analysis must also determine how conflicting accesses interleave in time. The number of invalidations and thus the amount of actual communication incurred by a group of conflicting accesses depends on how the accesses interleave. For instance, if two processors each perform two writes to the same location, then one of the processors will incur either one or two cache misses depending on whether one or more of the other processor's writes intercede between its own two writes. In this section, we show that both analyses, identifying conflicting accesses and determining how they interleave, are simpler to perform for software DSMs that support a release consistent shared memory model.

Analysis of coherence misses is significantly simplified by the delayed coherence property exhibited by most software implementations of release consistent (RC) memory consistency models. In RC models, maintaining coherence for individual shared memory accesses is delayed until special points specified explicitly by the application (see Section 2.2.2 of Chapter 2 for a discussion on delayed update techniques). For example, in MGS, which implements eager RC, coherence can only happen when the application executes a release operation.

The delayed coherence property is crucial from the standpoint of analysis for two reasons. First, individual shared memory accesses cannot generate invalidations[4]. Because of delayed coherence, individual shared memory updates are locally buffered and made visible to other processors only at release points. Therefore, the granularity at which coherence operations can interleave is increased from every single shared memory write (hardware DSMs) to release points (software DSMs). Because releases are less frequent than shared memory writes, the number of points in a program where coherence misses can occur is greatly reduced, thus simplifying analysis. Second, these coherence points in a program can be exactly identified by examining the program source code. RC memory models place the onus of coherence management on the programmer. Properly written shared memory programs for RC memory models include source-level annotations that

[4]This is not strictly true for MGS because of the Single-Writer mechanism. In MGS, a shared memory access to a remote page under Single-Writer mode will invalidate the remote page. This is the mechanism that reverts Single-Writer pages back to a normal level of coherence. This does not, however, complicate the analysis and will be discussed later in the section.

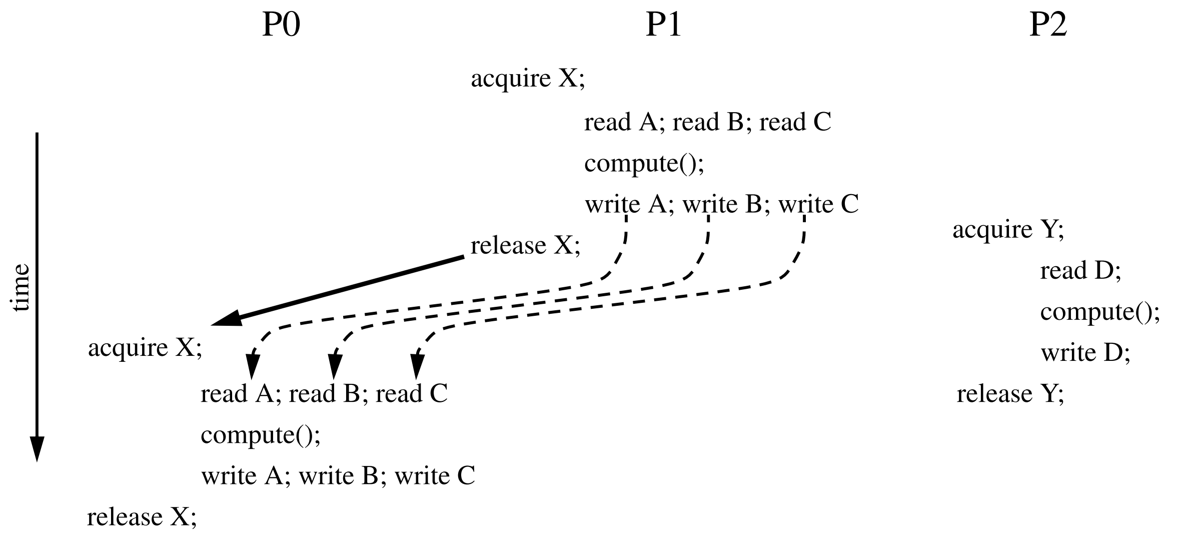


Figure 7.2: For most RC programs, the synchronization dependence information can be viewed as a summary of the data dependence information. The dotted lines indicate data dependences, and the solid line indicates a synchronization dependence.

identify release points [25]. Consequently, there is no mystery behind when coherence happens. Analysis can identify all coherence points by simply looking at the application's source code.

Despite the simplifications provided by the delayed coherence property, analysis must still address the following problem: identify those releases that generate communication, and for each such release, determine how much communication occurs. At first glance, this problem only looks slightly easier than the analysis problem encountered for hardware DSMs. Although there are fewer points where coherence can occur due to the coarser interleave granularity discussed above, it is still necessary to determine which processors share the pages that have been updated by the local processor performing the release operation (the invalidation set) in order to compute which pages are invalidated at any particular coherence point. Such analysis requires looking at all shared memory references performed across processors to come up with a set of conflicting pages.

Synchronization Analysis

The analysis to determine the conflict set for each coherence operation can be greatly simplified if we assume that applications use different synchronization variables to perform coherence operations on unrelated data. Figure 7.2 illustrates the behavior of an application that obeys this assumption. The figure shows three processors making mutually exclusive accesses to four shared memory locations named A, B, C, and D, using synchronization variables named X and Y. In this example, modifications to locations A, B, and C are always performed together, and use synchronization variable X. Modifications to location D are performed separately and use synchronization variable Y.

Notice that because different synchronization variables are used, processor $P2$ can perform computation associated with location D simultaneously with computation on the other three shared memory locations. In contrast, processors $P0$ and $P1$ must serialize their computations on locations A, B, and C because by convention, they use the same synchronization variable X. From a program correctness standpoint, the serialization is necessary to enforce mutual exclusion on the shared locations.

Figure 7.2 shows that for applications whose coherence points are annotated, *i.e.* the application includes acquire and release annotations as required by the RC memory model, each synchronization dependence between two processors represented as a *release* → *acquire* dependence over the same synchronization variable signifies data sharing. In our example, processors $P0$ and $P1$ share locations A, B, and C. This sharing is marked by the *release* → *acquire* dependence over the synchronization variable X from $P1$ to $P0$. Conversely, because there is no data sharing between processor $P2$ and processors $P0$ and $P1$, no *release* → *acquire* dependence can be identified between these processors due to the use of separate synchronization variables. The connection between data sharing and synchronization dependence solves part of our analysis problem. We no longer have to examine all shared memory references performed across processors to identify which releases generate communication. Instead, we can simply look for synchronization dependences.

While *release* → *acquire* synchronization dependences identify data sharing and thus identify where communication occurs, they do not reveal the volume of data communicated per *release* → *acquire* dependence. For instance, the synchronization dependence on variable X in Figure 7.2 actually represents the communication of three shared memory locations. Determining the volume of data communicated by each synchronization dependence requires analyzing data access information[5]. One approach is to pessimistically assume that all the data updated by code between an acquire and a release are communicated across the subsequent synchronization dependence. This assumption is true when the amount of data protected by each synchronization variable is always communicated across a synchronization dependence, a condition that holds for all the applications we studied in this thesis. Using this assumption, the volume of data communicated at the release point can be determined by analyzing the shared memory references performed inside the code between the acquire and the release to determine the number of unique pages touched. Notice that while such analysis involves examining data access information, the analysis is purely local in that it involves the references performed only by a single processor.

Before leaving our introduction of synchronization analysis, we must mention one limitation–synchronization analysis cannot handle false sharing (see Section 2.2.2 of Chapter 2 for a definition of false sharing). Communication caused by false sharing

[5]The volume of data communicated at each *release* → *acquire* synchronization dependence constitutes only part of what is computed by our analysis. Our analysis also computes the volume of several other events generated by the MGS protocol state machines in order to affect coherence on the data being moved across the synchronization dependence. More details on this topic appear in Section 7.1.3.

goes undetected in our analysis. The problem is that synchronization information only identifies true data sharing since synchronization is inserted into a program only when processors are accessing the same shared memory locations. Because there are no synchronization dependences associated with false data sharing, synchronization-based techniques cannot identify communication that arises from false sharing.

Clustering Analysis

Synchronization analysis applies to page-based software shared memory systems in general. The only requirement is that the shared memory supports a release consistent memory model that exhibits the delayed coherence property. Therefore, we can apply synchronization analysis to analyze the performance of most conventional software DSM systems. However, we can also use synchronization analysis to reason about DSSMPs as well if we incorporate the effects of clustering.

The extension for clustering in synchronization analysis is very simple. In DSSMPs, sharing between processors inside a single SSMP is handled by hardware mechanisms and thus bypasses software shared memory. Only sharing across SSMPs invokes software. This implies that not all dependences identified by synchronization analysis generate software-supported communication in DSSMPs. Instead, only those synchronization dependences that cross SSMP boundaries generate communication. Therefore, the extension for the analysis for DSSMPs involves an additional analysis phase. After synchronization dependences have been identified, a synchronization dependence graph is constructed. Each node in such a graph represents the code between an instance of an acquire and a release. The edges in a synchronization dependence graph represent *release* $\rightarrow$ *acquire* dependences as described above. Once constructed, the synchronization dependence graph is partitioned such that all the nodes in the graph that are executed by processors in the same SSMP are placed in the same partition. From the partitioning, analysis can identify *release* $\rightarrow$ *acquire* dependences that cause communication–they are those synchronization dependences which cross partition boundaries.

Figure 7.3 shows a partitioning example. The figure shows a synchronization dependence graph from a hypothetical application. The arrows with filled arrowheads represent synchronization dependences, while the arrows with unfilled arrowheads represent control dependences for graph nodes executed on the same processor. In this particular example, there are four processors, $P0$ through $P3$, organized across two SSMP nodes of two processors each. The dotted line represents both the physical SSMP node boundary for the four processors as well as the partition boundary for the graph nodes. Of the seven synchronization dependence arcs in the example graph, only three of the arcs cross SSMP node boundaries. The analysis will identify these three arcs as the communication-generating arcs. The other four arcs are "hidden" from the software shared memory layer and thus do not generate communication.

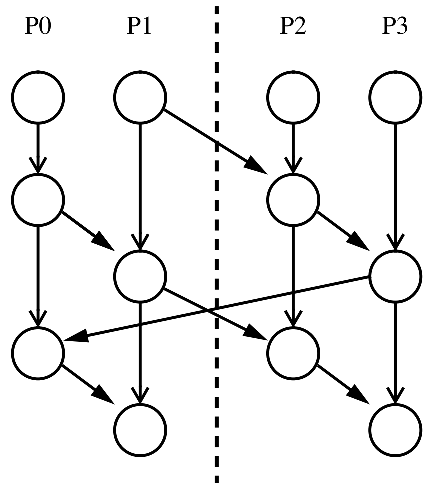


Figure 7.3: Analyzing clustering involves identifying synchronization dependences that cross SSMP node boundaries.

Discussion

Synchronization analysis provides a new opportunity to analyze communication in shared memory programs written for software shared memory systems that support a release consistent shared memory model. Our approach is based on the premise that race-free parallel programs use explicit synchronization whenever processors share data. This assumption allows our technique to infer data dependence information by analyzing synchronization dependences. This represents a shift from more traditional data-centric program analysis to an approach that is synchronization centric.

The primary benefit of synchronization-centric analysis over data-centric analysis is that synchronization dependence information is typically specified at a coarser granularity than data dependence information. Synchronization dependence graphs can be viewed as summaries of data dependence information in which multiple (but related) data dependence arcs are bundled into a single synchronization dependence arc. Because the graphs are smaller, analysis of synchronization dependence graphs involves lower complexity.

Notice our analysis does not completely discard information related to data objects. Data access information is still needed to derive communication volume. However, this information can be acquired through purely local analysis of each processor's reference stream. Global information about dependences that couples the accesses of multiple processors and gives rise to coherence misses is extracted from the synchronization dependence graph. Therefore, our analysis never constructs a data dependence graph.

There is still one problem that needs to be addressed. Building a synchronization dependence graph as described above for clustering analysis assumes that a particular ordering of graph nodes exists. In actuality, there may be many orderings that are legal. For example, consider a shared memory application that uses locking to perform atomic read-modify-write operations on a set of shared data objects. If the operations are commutative, then any ordering is legal, and the actual ordering that occurs at runtime depends on how different processors' attempts to acquire a particular lock interleave in time. Once the interleaving is fixed, the synchronization dependence graph can be constructed, but how a particular interleaving is chosen from a set of legal interleavings may significantly impact the communication analysis. Currently, we do not have a general solution to this problem. In Section 7.2.1, we solve the problem for the particular application used in our scalability study by optimistically assuming a maximally parallel schedule of synchronization operations, and then computing an interleaving based on the schedule. The technique proves to be accurate, but requires an analysis of the control flow of the code which we performed by hand. We believe it is possible to generalize the technique for some control structures (such as loops with static bounds), but that is beyond the scope of this thesis.

7.1.3 Performance Model

In the previous section, we introduced synchronization analysis. Synchronization analysis allows us to identify instances in a program where coherence communication occurs. Coupled with data access information, we can also determine how much data is communicated per communication-generating instance. This section applies the communication volume information provided by synchronization analysis in a performance model that predicts the execution time of an application on MGS.

Our model determines the impact of clustering on performance in a multigrain shared memory system. It assumes that parallel running time on an unclustered (all-hardware DSM) system is known. By using the communication volume information provided by synchronization analysis, the model predicts the overhead introduced by the software shared memory layer in terms of the cost of waiting for shared memory operations (latency), the cost of processing to run the protocol machines described in Section 4.2 of Chapter 4 (occupancy), and the cost of synchronization. This aggregate overhead introduced by software shared memory is then used to dilate the unclustered parallel running time to yield a prediction of execution time on the target clustered MGS system.

Figure 7.4 illustrates the main components in our performance model. The model consists of three major modules: the application description module, the machine description module, and the analysis engine. The rest of this section describes each of these modules in greater detail.

Application Description

The application description module specifies a set of application parameters to the analysis engine that describes the behavior of the application on an MGS system. As indicated by Figure 7.4, there are three types of application description parameters. *Parallel Runtime* specifies the time required to execute the application on an unclustered all-hardware DSM whose machine size (number of processors) equals the machine size of the target MGS system. In general, this value is difficult to predict. We expect the value to be

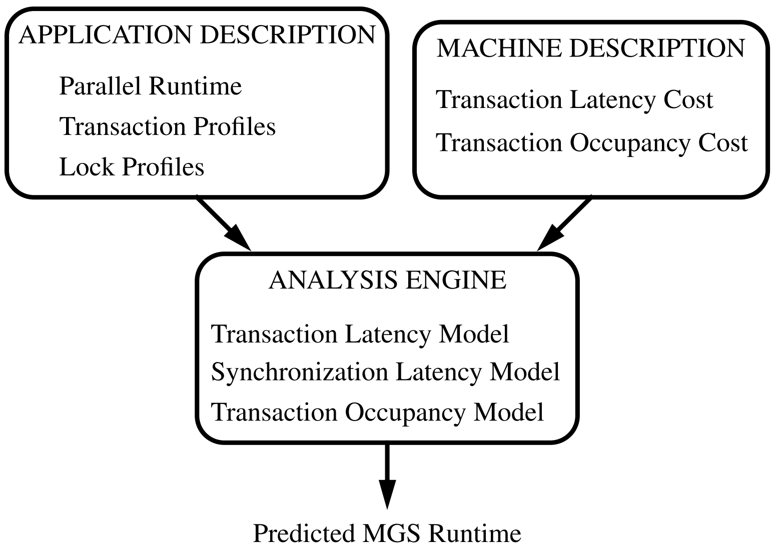


Figure 7.4: The performance analysis framework for multigrain systems.

provided to the model by measurement. The value can either be directly measured on a hardware DSM with the desired number of processors, or in those cases where this is not feasible (for instance, when the target MGS system is too large), the value can be extrapolated to the desired machine size from speedup curves obtained on a smaller hardware DSM.

Transaction Profiles are a set of parameters that specify the volume of software shared memory events incurred to support coherence misses. More specifically, the transaction profiles count the number of times the Local-Client, Remote-Client, and Server machines are invoked during the execution of an application, broken down into the different types of software shared memory events that invoke these machines. The transaction profiles are derived through synchronization analysis (see discussion below).

Table 7.1 lists the transaction profile parameters, categorized by the four types of shared memory transactions in MGS (see Section 4.2.1 in Chapter 4 for a description of these transaction types). All the parameters are listed under two columns, one for events that invoke the Local-Client machine (Local-Client Profiles), and another column for events that invoke the Remote-Client machine (Remote-Client Profiles). Profiles for Server machine events can be derived from the Local-Client profiles; therefore, separate parameters are not defined since they would be redundant.

The Local-Client Profiles consist of five parameters. `tlb_fault_counts`, `fetch_counts`, and `upgrade_counts` specify the number of TLB faults, page faults (including those that revert Single-Writer pages back to a normal level of coherence)[6], and upgrade faults, respectively. `release_counts` specifies the total number of releases performed, while `swrite_trans_counts` specifies the number of releases that result in transition to Single-Writer mode. The Remote-Client Profiles consist of two parameters that count events

[6]We do not attempt to distinguish between read and write fault types

Type	Local-Client Profiles	Remote-Client Profiles
TLB Fault	tlb_fault_counts	
Page Fault	fetch_counts	
Page Upgrade	upgrade_counts	
Release	release_counts swrite_trans_counts	inv_counts
Single-Writer		swrite_inv_counts

Table 7.1: Application transaction profile parameters.

pertaining to invalidation. `inv_counts` tracks invalidations due to release operations, and `swrite_inv_counts` tracks invalidations due to page faults that revert Single-Writer pages back to a normal level of coherence[7].

The synchronization analysis necessary to derive the transaction profile parameters involves a two-step process. First, a synchronization dependence graph is constructed and partitioned to yield a count of *release* $\rightarrow$ *acquire* synchronization dependences that generate communication, as described by the discussion on clustering analysis in Section 7.1.2. Second, volumes for all the transaction profile parameters in Table 7.1 are derived for a single *release* $\rightarrow$ *acquire* dependence. To do this, we analyze the shared memory references performed in the code prior to a synchronization dependence. As discussed earlier (see page 151), the goal of this analysis is to identify the number of unique pages touched. While this yields the volume of data communicated across a single synchronization dependence, our analysis must also determine the type and number of shared memory events generated by the MGS protocol state machines to affect the data movement. These events correspond exactly to the transaction profile parameters we seek. This two-step analysis can be better understood through a concrete example. In Section 7.2.1, we derive the transaction profile parameters for a specific application used to study the scalability of the MGS system.

Finally, *Lock Profiles* are a set of application description parameters that describe behavior associated with locks. These parameters only apply to applications that perform locking. In total, there are three parameters that constitute the lock profile; these parameters are listed in Table 7.2. `num_locks` is the total number of unique lock variables in the application. `lock_acquire_counts` is the total number of acquires performed across all the locks dynamically during execution. And `critical_section_time` specifies the average time spent inside a critical section. The first two parameters are determined through direct examination of the application source code. The last parameter can be derived using the transaction profile parameters in Table 7.1 (see Section 7.2.1 for more discussion). All three parameters are used by the Analysis Engine, described later in this section, to account for lock contention effects due to critical section dilation (see page 125 for an explanation of critical section dilation).

[7]Again, we do not distinguish between invalidation of read and write pages.

num_locks
lock_acquire_counts
critical_section_time

Table 7.2: Lock profile parameters.

Type	Latency	Mem Occ	Cli Occ
TLB Fault	tlb_fault_lat		
Page Fault	fetch_lat	fetch_occ	
Page Upgrade	upgrade_lat	upgrade_occ	
Release	release_lat swrite_trans_lat	release_occ swrite_trans_occ	inv_occ
Single-Writer	fetch_swrite_lat	swrite_occ	swrite_inv_occ

Table 7.3: Machine description parameters.

Machine Description

The machine description module specifies the performance of the underlying MGS system to the analysis engine. A set of machine parameters, listed in Table 7.3, describes the cost incurred on an MGS system for each of the software shared memory events listed in Table 7.1. Like Table 7.1, the costs have been categorized by the four types of shared memory transactions.

Two types of cost are accounted for–latency and occupancy. Latency cost parameters specify for how long a processor is stalled when performing a particular shared memory transaction that invokes software. These parameters appear in the "Latency" column of Table 7.3. Each event under the Local-Client Profile column in Table 7.1 has a latency cost associated with it listed in Table 7.3. In addition, Table 7.3 also lists a latency parameter called `fetch_swrite_lat`. `fetch_swrite_lat` is the cost of a page fault that reverts a page in the Single-Writer mode back to normal coherence. The number of such page faults is not a parameter in Table 7.1, but can be computed by subtracting `swrite_inv_counts` from `fetch_counts`.

Occupancy cost parameters specify the cost of processing handlers associated with software shared memory transactions. These parameters are listed under two columns in Table 7.3, "Mem Occ" and "Cli Occ," corresponding to the handler costs incurred on the Server and Remote-Client machines, respectively. There is a Remote-Client occupancy cost parameter corresponding to each event under the Remote-Client Profiles column in Table 7.1. In addition, there is a Server occupancy cost parameter for every event in the Local-Client Profiles column in Table 7.1 that requires service from the Server machine.

For the scalability study presented in Section 7.2, the machine description parameters in Table 7.3 are calibrated against our MGS prototype by using the measurements of software shared memory costs that appear in Table 6.3 on page 107. Since our analysis does not distinguish between reads and writes, we take the average from the "Load" and "Store" types in Table 6.3 whenever appropriate.

Analysis Engine

The analysis engine module predicts the execution time of an application, described by the parameters in the application description module, running on an MGS system, described by the parameters in the machine description module. Three performance models are used by the analysis engine in order to predict execution time from the application and machine specifications: the transaction latency model, the synchronization latency model, and the transaction occupancy model.

The analysis engine assumes that the runtime of an application on an MGS system is equal to the runtime on an all-hardware DSM of the same size (in total number of processors) dilated by three non-overlapping sources of overhead that occur in the MGS system, but that do not occur in the all-hardware system: stall due to software shared memory operations, stall due to contention on synchronization operations, and software shared memory protocol processing that interrupts (occupies) useful computation. Each of these overheads is computed by one of the three performance models in the analysis engine shown in Figure 7.4. The total dilation of the all-hardware DSM runtime is simply the sum of these three overheads since they are non-overlapping. Below, we describe the three performance models in detail.

The transaction latency model predicts the total software shared memory latency, Lat_{ssm}. This is the amount of time processors spend stalled on software shared memory transactions. Lat_{ssm} is simply the sum of all the Local-Client profile parameters from Table 7.1 weighted by the corresponding latency cost parameters reported in Table 7.3:

$$\begin{aligned}
Lat_{ssm} \;=\;& (tlb_fault_counts)(tlb_fault_lat) + \\
& (fetch_counts)(fetch_lat) + \\
& (upgrade_counts)(upgrade_lat) + \\
& (release_counts)(release_lat) + \\
& (swrite_trans_counts)(release_swrite_lat) + \\
& (fetch_counts - swrite_inv_counts)(fetch_swrite_lat) \qquad (7.1)
\end{aligned}$$

The synchronization latency model predicts the stall time suffered by processors due to lock contention. As we saw in Chapter 6, lock contention is certainly severe in applications that spend practically all of their time performing locking, such as TSP. Surprisingly, lock contention also has a significant performance impact on applications that perform only modest amounts of locking. For instance, when we first developed our performance framework, we used the Water application as a benchmark to validate our performance model. Early versions of the model systematically under-predicted runtime by as much as 40%. It wasn't until we took a closer look at our performance results that we realized this discrepancy was due to lock contention. This was a surprising result because Water is careful about how it performs locking. It uses a very large synchronization space (one lock per molecule in the simulation) to distribute the locking load in order to reduce contention.

The reason why lock contention can be so severe even for applications that do not use centralized locks is because systems that rely on software shared memory suffer from the critical section dilation effect discussed in Section 6.3.2 of Chapter 6. Critical section dilation introduces large software shared memory overheads during the time when a processor has acquired a lock. This significantly increases the probability that another processor will try to acquire the lock before it has been relinquished, and thus increases lock contention. Any performance model for software shared memory systems must account for lock contention; otherwise, they cannot accurately predict performance for applications that are vulnerable to critical section dilation.

To account for lock contention, we use a simple closed queuing network model. In order to facilitate a lucid exposition of the analysis, we first assume a single lock, and extend the analysis to handle multiple locks later in this section. The queuing network used to model lock contention is shown in Figure 7.5. It consists of two queues, an $M/M/1$ queue (Queue 0) and an $M/M/s$ queue (Queue 1) where $s = P$, the total number of processors in the system[8] (see [39] for more details on these queues and their analysis).

In this queuing network, customers represent processors, and the queues represent two different processor activities. A customer entering Queue 0 signifies a processor trying to acquire the lock. If the queue is empty, then the customer enters the server immediately, corresponding to a successful lock acquire. If however the queue is busy, then the customer must wait. This is the lock contention case. The customer remains in Queue 0 for as long as the processor holds the lock (*i.e.* the time spent in the critical section). A customer leaving Queue 0 signifies a release of the lock. Notice that since Queue 0 only has one server, the mutual exclusion property of the lock is enforced. When a customer is in Queue 1, the corresponding processor is performing parallel computation. Since Queue 1 has P servers, there is always an available server for a customer; therefore, all processors can be performing parallel computation at the same time. In this simple model, processors alternate between doing work in parallel, and performing lock operations that contend whenever two or more lock operations occur simultaneously.

λ and μ are the service rates for Queue 1 and Queue 0, respectively. They parameterize the distributions that govern the amount of time customers spend in each queue. For simplicity, we assume both of these distributions are Poisson. λ corresponds to the average rate at which processors perform lock operations. This rate is equal to the average number of lock acquires performed by each processor divided by the parallel runtime of the application. μ corresponds to the average rate at which processors complete critical sections once they have acquired the lock. This rate is equal to the inverse of the average number of cycles a processor spends inside a critical section, or the `critical_section_time` parameter from Table 7.2.

The closed queuing network in Figure 7.5 is known as a *Jackson network*. The

[8] As is explained, Queue 1 has as many servers as customers, so it is impossible for any queuing to occur; therefore, we have omitted drawing the queue itself in Figure 7.5. This kind of queue is known as a delay server.

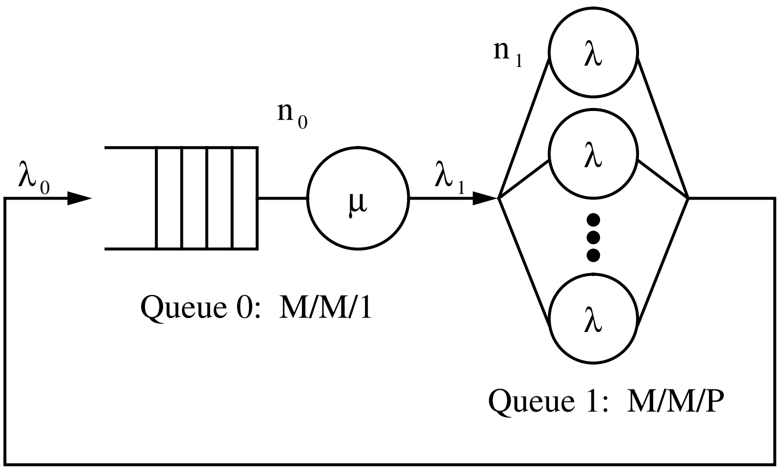


Figure 7.5: Closed queuing system used to model lock contention due to critical section dilation.

probability that n_0 customers are in Queue 0 and n_1 customers are in Queue 1, where $n_0 + n_1 = P$, is given by:

$$p(n_0, n_1) = \frac{1}{G(P)} \left(\frac{\lambda_0}{\mu}\right)^{n_0} \left(\frac{\lambda_1^{n_1}}{\prod_{i=1}^{n_1} \lambda \min(i, P)}\right) \tag{7.2}$$

where λ_0 and λ_1 are the solutions satisfying the flow equations between Queue 0 and Queue 1, respectively. Since the queuing network is a closed system of two queues, the flow equations can be satisfied trivially. Choosing the solution $\lambda_0 = \lambda_1 = 1$, we have:

$$\begin{aligned} p(n_0, n_1) &= \frac{1}{G(P)} \left(\frac{1}{\mu}\right)^{n_0} \left(\frac{1}{\prod_{i=1}^{n_1} \lambda \min(i, P)}\right) \\ &= \frac{1}{G(P)} \left(\frac{1}{\mu}\right)^{n_0} \left(\frac{1}{\lambda}\right)^{n_1} \frac{1}{n_1!} \end{aligned} \tag{7.3}$$

$G(P)$ in Equation 7.3 is a normalizing constant and can be expressed as:

$$G(P) = \sum_{n_0+n_1=P} \left(\frac{1}{\mu}\right)^{n_0} \left(\frac{1}{\lambda}\right)^{n_1} \frac{1}{n_1!} \tag{7.4}$$

The queuing network in Figure 7.5, and its solution, Equation 7.3, allows us to compute synchronization latency due to lock contention. The lock acquire latency per lock experienced by each processor is equal to the average time a customer spends waiting in Queue 1. This wait time is the expected queue length at Queue 1 multiplied by the average service time per customer, μ^{-1}. If we multiply by the number of lock acquires performed by each processor, we can compute the total synchronization latency, Lat_{syn}:

$$\begin{aligned} Lat_{syn} &= \left(\frac{lock\_acquire\_counts}{P}\right)\left(\frac{1}{\mu}\right)E[n_1] \\ &= \left(\frac{lock\_acquire\_counts}{P}\right)\left(\frac{1}{\mu}\right)\sum_{n_1=0}^{P} n_1 p(n_1, P-n_1) \end{aligned} \tag{7.5}$$

where $lock\_acquire\_counts$ is the total number of lock acquire operations performed, from Table 7.2.

Equation 7.5 provides the synchronization latency solution assuming a single lock. Most applications, however, employ multiple locks. The general problem of contention with multiple locks is difficult to model. The naive solution would solve Equation 7.5 once for each unique lock in the system, and then sum the individual latencies. Unfortunately, this solution is incorrect. Consider the solution of a single lock in a multi-lock application. In this case, Queue 1 in Figure 7.5 not only models parallel computation, but it must also model the latency introduced by all other locks. The latencies computed for all other locks must feed back into the rate parameter for Queue 1, λ. Similarly, the solution for the single lock in question must also feed back into all other locks. Therefore, the correct solution in the multi-lock case requires simultaneous solutions for all locks in a self-consistent fashion. If the number of locks in the application (`num_locks` from Table 7.2) is large, this computation becomes intractable.

We can greatly simplify the multi-lock case if we assume that all locks have the same lock profiles. In other words, processors access all locks in the application homogeneously, and the amount of critical section dilation is equal for all locks. Under this assumption, we expect the solution to Equation 7.5 to be identical for all the locks. This makes the feed back very easy to model. Specifically, we can model the feed back by using a new rate for Queue 1, λ', that is computed as:

$$\lambda' = \left(\frac{1}{\lambda} + (num\_locks)(Lat_{syn})\right)^{-1} \tag{7.6}$$

The solution to Equation 7.5 is a bit harder using λ' as the rate parameter for Queue 1 since any correct solution must satisfy Equations 7.3, 7.5, and 7.6 simultaneously. However, this problem is tractable and can be solved using an iterative method.

The last model in the analysis engine's arsenal is the transaction occupancy model. This model accounts for the overhead of processing software shared memory handlers (on the Remote-Client and Server machines) that "occupy" processors, thus impeding their progress on useful computation. Like its latency counterpart, Lat_{ssm}, the amount of raw occupancy, Occ_{raw}, can be computed by summing over all Remote-Client and Server occupancy counts in Table 7.1 weighted by the corresponding occupancy costs reported in Table 7.3. Notice that while Remote-Client occupancy profiles are specified explicitly in Table 7.1 (last column), Server occupancy profiles are not. The Server occupancy counts are identical to the Local-Client profiles for those transactions which incur Server occupancy (*i.e.* transactions that have a non-empty entry in column 3 of Table 7.3). The raw occupancy can be expressed as:

$$\begin{aligned} Occ_{raw} = \; & (fetch\_counts)(fetch\_occ) + \\ & (upgrade\_counts)(upgrade\_occ) + \\ & (release\_counts)(release\_occ) + \\ & (swrite\_trans\_counts)(swrite\_trans\_occ) + \\ & (fetch\_counts - swrite\_inv\_counts)(swrite\_occ) + \\ & (inv\_counts)(inv\_occ) + \\ & (swrite\_inv\_counts)(swrite\_inv\_occ) \end{aligned} \tag{7.7}$$

Using Equation 7.7 as the total occupancy overhead would be pessimistic because when a software shared memory handler occupies a processor, it slows the processor down only if the processor was doing useful work. If the processor was idle, for instance waiting on a shared memory transaction or a synchronization operation, then the cost of the occupancy would be "hidden." Therefore, we should only charge the cost of those handlers that occupy useful computation. To model this effect, we assume that handlers either occupy a processor during useful computation, or are completely hidden by shared memory or synchronization latency. We do not account for partially hidden handler costs. This case arises, for example, when a handler is initiated during a page fault transaction, and part-way through the handler, the transaction completes. Furthermore, we assume the probability that a handler occupies useful work is proportional to the fraction of time that processors spend doing useful work. The actual occupancy cost, Occ_{act}, can be expressed as:

$$Occ_{act} = \left(\frac{R + Occ_{act}}{R + Occ_{act} + W} \right) Occ_{raw} \tag{7.8}$$

where R is the parallel running time of the application on a hardware DSM, and $W = Lat_{ssm} + Lat_{syn}$. Notice that in Equation 7.8, we use $R + Occ_{act}$ in both the numerator and denominator instead of just R. This is because as we dilate R with occupancy overhead, the probability that handlers will occupy useful work increases. Equation 7.8 is quadratic in Occ_{act}. Solving for Occ_{act} (taking the positive root), we have:

$$Occ_{act} = -\frac{(R + W - Occ_{raw}) + \sqrt{(R + W - Occ_{raw})^2 + 4ROcc_{raw}}}{2} \tag{7.9}$$

Combining Equations 7.1, 7.5, and 7.9, we arrive at the prediction of runtime on the target clustered MGS system:

$$PredictedMGSRuntime = R + Lat_{ssm} + Lat_{syn} + Occ_{act} \tag{7.10}$$

7.2 Scalability Study

In this section, we apply the analytical framework presented in Section 7.1 to study the scalability of MGS. First, in Section 7.2.1, we describe the application we use for the scalability study, called Water-Kernel-NS, and derive the application description parameters for Water-Kernel-NS required by the model. Then in Section 7.2.2, we validate the accuracy of our model by comparing model predictions of Water-Kernel-NS execution time with experimental execution times observed on the MGS prototype. Finally, in Section 7.2.3, we use the validated model to study MGS performance when both problem size and machine size are scaled.

7.2.1 Application Description for Water

Our scalability study of the MGS system uses the Water-Kernel-NS application. Water-Kernel-NS is a kernel of the Water application from SPLASH consisting of the force interaction computation loop, as shown in Figure 6.9 on page 124. A summary of the problem sizes we use for Water-Kernel-NS appears in Table 6.4 on page 113, and a summary of the sequential and parallel performance of Water-Kernel-NS appears in Table 6.5 on page 114.

Water-Kernel-NS is almost identical to the Water-Kernel code described in Section 6.4.1 of Chapter 6. The only difference is in the choice of a compile-time constant called "CUTOFF." The CUTOFF constant specifies the maximum interaction separation between water molecules. Only for those pairs of molecules that are separated by a distance smaller than the CUTOFF constant is a force interaction computed; force interaction computation for molecules that exceed this separation are not considered. In Water-Kernel-NS, we choose a CUTOFF constant that is equal to the diameter of the simulation space; consequently, all possible N-Squared pairwise interactions are performed (thus the suffix "NS")[9]. Such a choice of the CUTOFF constant removes any data dependent behavior thus simplifying our analysis.

As described in Section 7.1.3, there are three components in the application description: parallel runtime, transaction profiles, and lock profiles. In the rest of this section, we describe the derivation of these application description parameters for Water-Kernel-NS.

Parallel Runtime

We measure the parallel runtime of Water-Kernel-NS on the Alewife machine. The measurement uses a problem size of 512 molecules, and a machine size of 32 processors. The result of this measurement appears in Table 6.5 on page 114.

Since our intention is to use our model to study scalability, it will be necessary to extrapolate the measured runtime to both larger problem sizes and machine sizes. Because the force interaction computation loop in Figure 6.9 is doubly nested, the runtime

[9] In Water and Water-Kernel, the CUTOFF constant is set to one half the diameter of the simulation space.

is quadratic in the problem size, or the number of molecules. Therefore, if we increase the problem size by scaling the number of molecules by a factor K, we must increase parallel runtime by a factor K^2. Scaling machine size is a bit more tricky because the efficiency of the application changes with machine size. For simplicity, we assume that runtime decreases linearly as we scale up machine size, *i.e.* we assume linear speedup from a machine size of 32 processors on up. In practice, this is a good assumption when problem size is large compared to machine size.

Transaction Profiles

Deriving transaction profiles for an application requires performing two types of analysis. First, we must determine the number of *release* $\rightarrow$ *acquire* dependences that cross SSMP node boundaries. Second, for each *release* $\rightarrow$ *acquire* dependence that crosses a SSMP node boundary, we must determine the values for all the transaction profile parameters listed in Table 7.1.

The total number of *release* $\rightarrow$ *acquire* pairs is equal to the total number of lock acquires performed, or N^2 (see discussion on Lock Profiles below). Of this total, some fraction will cross SSMP node boundaries. Computing this fraction exactly is difficult because it requires knowledge about how acquire operations interleave during an actual execution. For simplicity, we can *ignore the effects of interleaving* to create a naive model. In this naive model, we assume that the fraction of *release* $\rightarrow$ *acquire* pairs that cross SSMP node boundaries is equal to the fraction of remote acquires. A remote acquire occurs each time an acquire is performed on a lock associated with a molecule owned by a remote processor (a processor on a remote SSMP). The number of remote acquires performed by a single processor ignoring interleaving can be expressed as:

$$FractionRemoteAcquires = \frac{\left(\frac{N}{2} - \frac{N}{nSSMPs}\right)\frac{N}{P}}{\frac{N^2}{P}} \tag{7.11}$$

where $nSSMPs$ is the number of SSMPs, and P is the number of processors in the machine. The expression inside the parentheses in the numerator of Equation 7.11 is the number of remote acquires performed by a processor in a single iteration of the inner loop of the force interaction computation (see Figure 6.9). This expression is multiplied by the number of iterations of the outer loop to yield the total number of remote acquires. Dividing by the total number of acquires performed by a single processor, $\frac{N^2}{P}$, yields Equation 7.11.

For comparison, we consider another model that requires more analysis effort than Equation 7.11, but accounts for the dynamic interleaving of acquire operations. The analysis examines the iteration space of the force interaction computation, an example of which is shown in Figure 7.6. In Figure 7.6, the X-axis represents iterations of the inner loop, while the Y-axis represents iterations of the outer loop. The iteration space of the computation is the lower triangle drawn in solid lines in Figure 7.6. Furthermore, the figure shows how the iteration space has been partitioned amongst processors, assuming a machine with 8 processors.

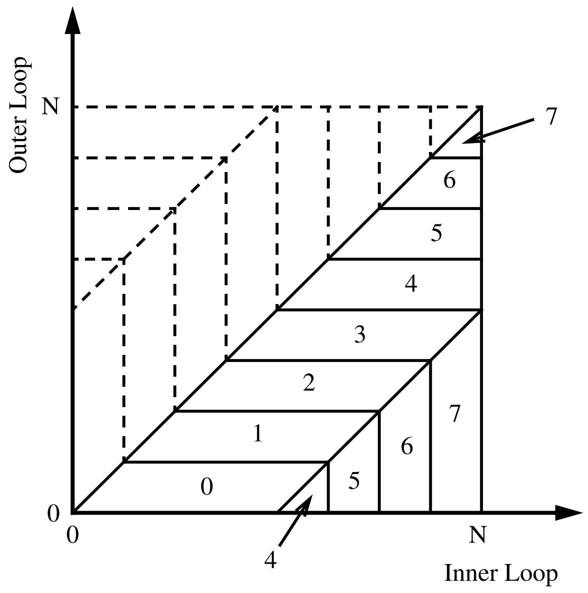


Figure 7.6: Partitioning of the iteration space of the force interaction computation in Water-Kernel-NS.

Each point in Figure 7.6 represents a single iteration of the force interaction computation. Equivalently, a point (i, j) represents an interaction between molecules i and j, and thus represents the acquisition and release of locks i and j. If we reflect the triangular iteration space in Figure 7.6 across the diagonal, indicated by the dotted lines in the figure, thus forming an NxN matrix, then all the acquires performed on lock i occur in the interactions in row i[10]. Consequently, given an interaction (i, j), the acquire of lock i will have a synchronization dependence with the release of lock i performed in some other interaction along row i. This synchronization dependence crosses SSMP node boundaries if the processor performing the other interaction resides on a remote SSMP. Notice a similar line of reasoning applies to the synchronization dependence associated with lock j.

To determine which interaction in row i is the source of the synchronization dependence for interaction (i, j) requires temporal information regarding when a particular iteration executes. We approximate temporal information in the following way. Each point in the iteration space receives a timestamp, where an iteration that receives timestamp t is the tth iteration performed by its local processor. Notice that because of symmetry, iteration (i, j) and its reflected iteration (j, i) in Figure 7.6 will be assigned the same timestamp. Then, we sort every row in the NxN matrix in Figure 7.6 according to timestamps. After the sort, adjacent iterations in the same row have a synchronization dependence, and that dependence crosses SSMP node boundaries if the two processors associated with the two iterations are on separate SSMPs. By looking at all iteration pairs

[10]Because of symmetry, the interactions in row i are identical to the interactions in column i, so we can look at either rows or columns.

that are row-wise adjacent, we can estimate the number of synchronization dependences that cross SSMP node boundaries.

Once the number of *release* $\rightarrow$ *acquire* synchronization dependences that cross SSMP node boundaries have been computed, we need to determine the values for the transaction profile parameters in Table 7.1 for each synchronization dependence. For the Water-Kernel-NS code, this analysis is relatively straight forward. From the standpoint of the software shared memory layer, each *release* $\rightarrow$ *acquire* dependence represents a coherence miss that migrates a single molecule object from one SSMP to another SSMP. Since the size of the molecule data structure is 672 bytes, each molecule fits inside a single page so that the migration involves only one page[11].

The migration of a molecule object is initiated by a page fault which pulls over a copy of the page to the requesting SSMP. Because accesses to molecule objects occur in an exclusive fashion (since the code uses locks to achieve mutual exclusion), it is likely that at the time of the page fault, there is exactly one outstanding copy of the page in question, and furthermore, this page is in Single-Writer mode. Hence, the page fault must revert this Single-Writer copy back to a normal level of coherence via invalidation. Next, our analysis must recognize that for each *release* $\rightarrow$ *acquire* dependence, the code is performing a read-modify-write operation. Therefore, the access that causes the initial page fault is a read, and a page upgrade fault occurs when the first write is performed. Finally, when the requesting processor completes all accesses to the object, it performs a release operation to make its modifications visible to all other processors. Again, because of the exclusive fashion in which accesses are performed, it is likely that at the time of the release, the requesting processor has the only copy of the page; consequently, the release transitions the page to the Single-Writer mode. The transaction profile parameters for each *release* $\rightarrow$ *acquire* synchronization dependence that crosses SSMP node boundaries is:

1 page fault
1 invalidation of Single-Writer copy
1 page upgrade fault
1 single-writer release

Notice that when a synchronization dependence is contained within an SSMP, no coherence actions occur. Since the last action described above transitions the page to the Single-Writer mode, all software shared memory mechanisms are "turned off" until the page reverts back to a normal level of coherence. This occurs on the next synchronization dependence that crosses SSMP node boundaries.

Lock Profiles

The lock profiles consist of three parameters, `num_locks`, `lock_acquire_counts`, and `critical_section_time`, as shown in Table 7.2. In Water-Kernel-NS, there is one lock

[11]While it is possible for a particular molecule to straddle two pages, for simplicity, we ignore this effect.

for each molecule, so `num_locks` equals N, the problem size, or the number of molecules in the simulation. The number of lock operations or acquires that occur dynamically during the execution is equal to twice the number of pairwise molecule interactions (each interaction updates both molecules involved in the interaction). Since there are a total of $\frac{N^2}{2}$ pairwise interactions, the number of lock acquires equals N^2.

The `critical_section_time` parameter specifies the average time spent inside each critical section. In Water-Kernel-NS, a critical section is very small–it consists of a single read-modify-write operation, as shown in Figure 6.9. However, the time spent inside the critical section becomes large because of critical section dilation. The dilation occurs due to the software shared memory overheads suffered in order to maintain coherence on the molecule data structure. The specific coherence events involved are exactly the ones derived for the transaction profile parameters. Therefore, the value of the `critical_section_time` parameter is simply the costs for each of these coherence events from Table 7.3:

$$\begin{aligned} critical\_section\_time &= fetch\_lat + fetch\_swrite\_lat + \\ &\quad upgrade\_lat + swrite\_trans\_lat \end{aligned} \tag{7.12}$$

Tiled Water-Kernel-NS

As part of our scalability study, in Sections 7.2.2 and 7.2.3 below, we also apply our analysis to a version of Water-Kernel-NS that has been tiled using the tiling transformation described in Section 6.4.1 of Chapter 6 for Water. Below, we briefly discuss an application description for Water-Kernel-NS under the tiling transformation.

The analysis to determine the application description parameters for the tiled version of Water-Kernel-NS is extremely simple because tiling enhances the locality of the force interaction computation loop shown in Figure 6.9. In particular, tiling removes all inter-SSMP sharing during the interaction of molecules inside a pair of tiles; sharing across SSMPs occurs only when two new tiles are selected. The enhanced locality provided by tiling enables two simplifications to the application description analysis. First, because locality significantly reduces sharing across SSMPs, it is rare for processors in separate SSMPs to acquire the same lock. Therefore, we can ignore lock contention. Second, *release* $\rightarrow$ *acquire* synchronization dependences cross SSMP node boundaries only when new tiles are selected. Furthermore, when two new tiles are selected by an SSMP, all molecules in those tiles are moved to the requesting SSMP once and remain cached until two new tiles are selected. Therefore, the number of *release* $\rightarrow$ *acquire* synchronization dependences that cross SSMP node boundaries is:

$$\left(\frac{N}{tile\_size}\right)^2 (2tile\_size) \tag{7.13}$$

where *tile_size* is the number of molecules per tile. The left multiplier in Equation 7.13 is the number of pairings of tiles, and the right multiplier is the number of molecules moved per pairing of tiles. For each *release* $\rightarrow$ *acquire* synchronization dependence that

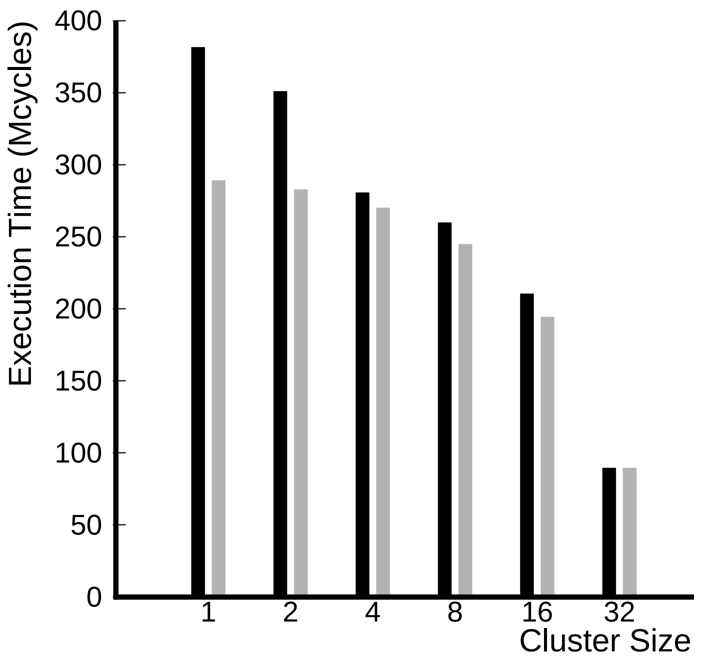


Figure 7.7: Validation for Water-Kernel-NS model ignoring interleaving effects.

crosses SSMP node boundaries, the same coherence events occur in the tiled version of Water-Kernel-NS as in the original version of Water-Kernel-NS.

7.2.2 Model Validation

In this section, we validate the accuracy of our runtime prediction based on the analysis described in Section 7.2.1 above. Our validation considers both the original and tiled versions of the Water-Kernel-NS code, and for the original code, we examine prediction accuracy using both the naive analysis that ignores the effects of interleaving acquire operations, and the more detailed analysis that accounts for interleaving.

Figure 7.7 shows the validation results for the naive analysis of the Water-Kernel-NS code. The figure shows the results of the model prediction (right set of bars) alongside measured performance on our MGS prototype with 32 processors (left set of bars). There are a set of two bars for each SSMP node size, which is varied from 1 to 32 processors in powers of two as was done for the results presented in Chapter 6. The measured and analytical bars for the 32 processor SSMP node size are identical because our model does not predict hardware DSM performance.

The agreement between the experimental and analytic numbers in Figure 7.7 is decent (average error: -12.2%). Overall, the model under-predicts runtime. This is because the naive analysis optimistically assumes that all acquires performed on locks associated with molecules owned by the local SSMP node do not generate synchronization dependences

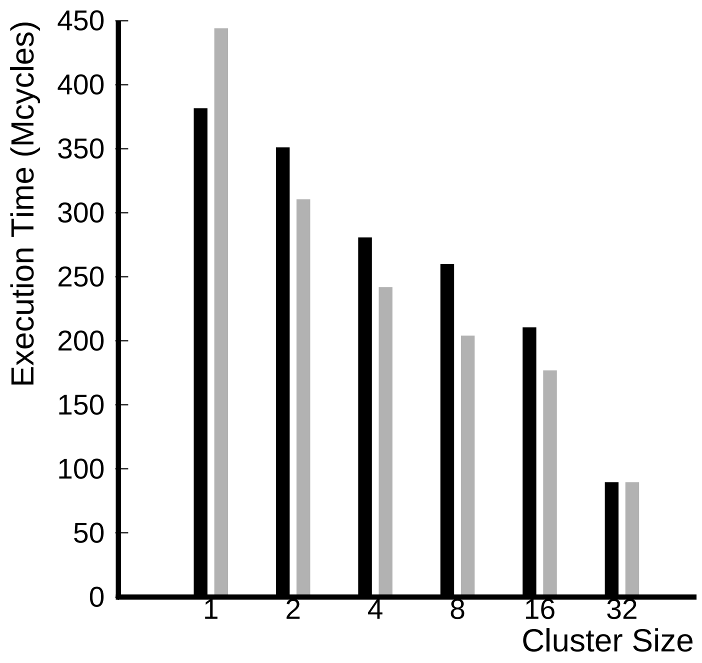


Figure 7.8: Validation for Water-Kernel-NS model accounting for interleaving effects.

that cross SSMP node boundaries. In actuality, this optimistic assumption is incorrect because these so-called "local" acquires interleave with remote acquires performed by processors on other SSMPs. While one can argue that the impact of this incorrect assumption is small as evidenced by the reasonable model agreement we observe, there is a more troubling consequence of the naive analysis: the shape of the predicted curve does not match the shape of the measured curve. The predicted curve is convex whereas the measured curve is more concave. This shape mismatch implies the naive analysis incorrectly predicts the sensitivity of application performance to SSMP node size.

Figure 7.8 shows the validation results for the detailed analysis of the Water-Kernel-NS code that accounts for interleaving of acquire operations. Notice the agreement is slightly better in Figure 7.8 (average error: -9.3%). More importantly, the shape of the predicted curve more closely matches the shape of the measured curve. In Section 7.2.3 where we study scalability, we will use the detailed analysis rather than the naive analysis.

Finally, Figure 7.9 shows the validation results for the analysis of the tiled Water-Kernel-NS code. The figure shows that there is excellent agreement between our prediction and the measured results (average error: -0.8%).

7.2.3 Scaling Results

This section presents the results of the scaling study using the performance model presented in Section 7.1.3. We present scaling results for the Water-Kernel-NS code, both in

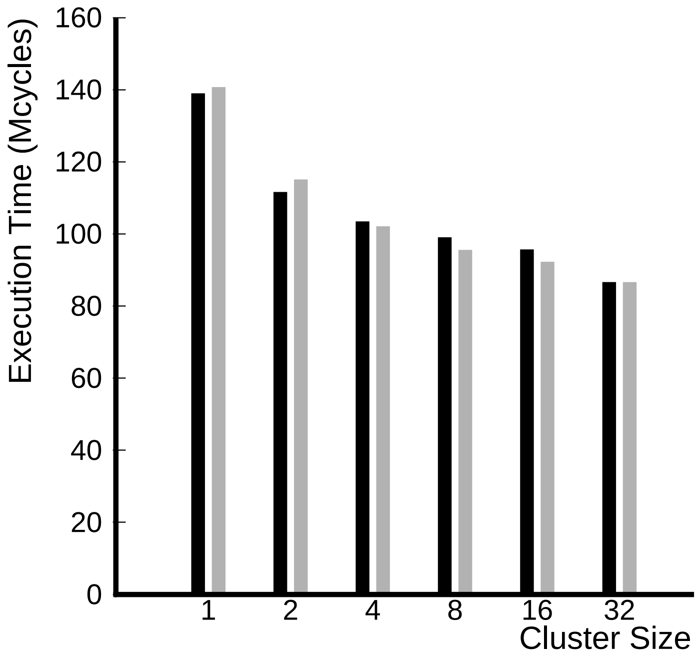


Figure 7.9: Validation for tiled Water-Kernel-NS model.

its original form, and with the tiling transformation. We use the application descriptions derived for both versions of Water-Kernel-NS in Section 7.2.1 to drive the performance model.

In the scaling study, we address three fundamental questions. First, what is the impact of scaling machine size on the performance framework parameters, the multigrain potential and the breakup penalty? Addressing this question allows us to extrapolate the experimental results observed in Chapter 6 to larger machines. Second, what is the impact of scaling machine size on the performance of DSSMPs built using small SSMP nodes? Can we achieve most of the multigrain potential at reasonably small SSMP nodes, as was observed for most applications in Chapter 6, or does scaling machine size require scaling SSMP node size at a similar rate to maintain a constant level of efficiency? And finally, what is the tradeoff between machine size and SSMP node size in maintaining high performance on large-scale machines? A specific performance requirement can be met by either building smaller machines using larger SSMP nodes, or larger machines using smaller SSMP nodes. Understanding the performance tradeoff between machine size and SSMP node size allows the architect of large-scale systems to evaluate the optimal point in the space of DSSMP configurations for a particular application.

Tables 7.4 and 7.5 summarize the results of the scaling study. Table 7.4 reports the predicted execution times, and Table 7.5 reports the corresponding speedups for the Water-Kernel-NS code in both its original and transformed versions. Each column of the two tables, except for the first column, reports a performance number at a specific

Procs	1	2	4	8	16	32	64	128	256	512
Water-Kernel-NS										
1	33963.0									
32	6475.3	4239.5	3106.5	2520.7	2177.3	1434.7				
64	3330.2	2194.9	1620.5	1324.3	1160.7	1047.2	717.4			
128	1779.2	1223.2	943.0	799.1	720.0	666.2	610.7	358.7		
256	870.1	607.4	474.6	406.0	368.0	341.8	314.2	271.2	179.3	
512	472.7	359.8	302.7	273.3	256.7	244.9	231.8	211.0	171.9	89.7
Tiled Water-Kernel-NS										
1	33760.0									
32	1634.3	1529.4	1476.6	1450.2	1436.9	1386.4				
64	921.0	817.1	764.7	738.3	725.1	718.5	693.2			
128	563.0	460.5	408.6	382.4	369.2	362.5	359.2	346.6		
256	382.4	281.5	230.3	204.3	191.2	184.6	181.3	179.6	173.3	
512	290.9	191.2	140.7	115.1	102.1	95.6	92.3	90.6	89.8	86.

Table 7.4: Scaling results summary–execution times. Each row corresponds to a machine size and each column corresponds to an SSMP size. All quantities are reported in millions of cycles.

SSMP node size; each row reports all the performance numbers at a specific machine size specified by the first column. We examine the results in Tables 7.4 and 7.5 in greater detail below. We first address the impact of scaling on the performance framework parameters and SSMP node size at a given machine size, then we examine the tradeoff between machine size and SSMP node size.

Performance Framework Results

We first present results for Water-Kernel-NS on 32, 128, and 512 processors using the same performance framework used to present the experimental results in Chapter 6. We examine performance framework results for both the original and tiled versions of the Water-Kernel-NS code.

A 512 processor DSSMP represents a factor of 16 increase in machine size as compared to the prototype used in our experimental study. In addition to scaling machine size, we also scale the problem size since a larger problem size more accurately represents the kind of workloads that will run on the larger machines. For our scalability study, we choose a problem size that represents a 16-fold increase in the amount of work as compared to the problem size used in our experimental study. Since, work grows quadratically with the number of simulated molecules in Water-Kernel-NS, we increase the number of molecules by a factor of 4, from 512 molecules to 2048 molecules.

Figures 7.10, 7.11, and 7.12 show the scaling results for 32, 128, and 512 processors, respectively, for the original version of Water-Kernel-NS. The multigrain potential and breakup penalty metrics in our performance framework are labeled at the bottom of each graph.

Procs	1	2	4	8	16	32	64	128	256	512
Water-Kernel-NS Speedups										
32	5.3	8.0	10.9	13.5	15.6	23.7				
64	10.2	15.5	21.0	25.7	29.3	32.4	47.3			
128	19.1	27.8	36.0	42.5	47.2	51.0	55.6	94.7		
256	39.0	55.9	71.6	83.7	92.3	99.4	108.1	125.3	189.4	
512	71.9	94.4	112.2	124.3	132.3	138.7	146.5	161.0	197.6	378.8
Tiled Water-Kernel-NS Speedups										
32	20.7	22.1	22.9	23.3	23.5	24.4				
64	36.7	41.3	44.2	45.7	46.6	47.0	48.7			
128	60.0	73.3	82.6	88.3	91.5	93.1	94.0	97.4		
256	88.3	119.9	146.6	165.3	176.6	182.9	186.2	188.0	194.8	
512	116.1	176.6	239.9	293.3	330.5	353.2	365.8	372.5	375.9	389.6

Table 7.5: Scaling results summary-speedups. Each row corresponds to a machine size and each column corresponds to an SSMP size.

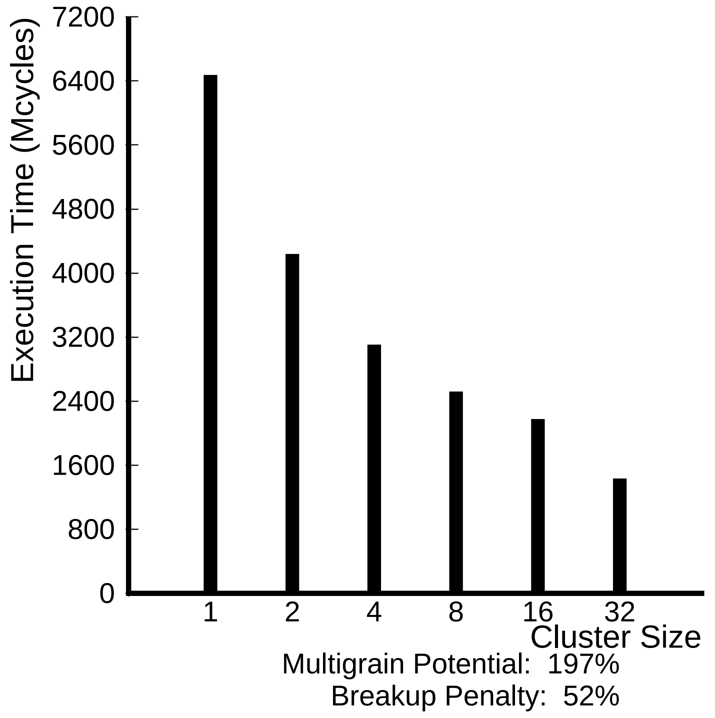


Figure 7.10: Scaling Water-Kernel-NS. 32 Processors.

Since the machine represented in Figure 7.10 is the same size as the machine used in the experimental study, 32 processors, we can observe the effects of scaling problem size from 512 molecules to 2048 molecules by comparing Figure 7.10 to the left set of bars in Figure 6.13 on page 134[12]. Such a comparison reveals surprisingly that there is not a significant difference in behavior even when problem size is increased by a factor of 16. In particular, the multigrain potential is still large (197%), and the breakup penalty is still significant (52%). Generally, increasing problem size increases the granularity of sharing. Therefore, we expect software shared memory architectures to be more competitive with hardware DSMs at these larger problems, and we expect the performance profile in Figure 7.10 to "flatten," thus resembling the curves in the "Easy" category described in Chapter 6. The reason this effect is not observed for Water-Kernel-NS (nor would it be observed for Water) is because the application exhibits poor data locality due to the all-to-all sharing pattern in the force interaction computation loop. Poor data locality prevents the granularity of sharing from increasing significantly as a result of an increase in problem size.

By looking at Figures 7.11 and 7.12, we observe that breakup penalty increases with increasing machine size. At 128 processors, the breakup penalty is 70% (up from 52% for 32 processors), and at 512 processors, it increases to 91%. Again, this is attributable to poor data locality. When machine size is increased, the time to perform computation associated with the application reduces proportionally–this is an assumption in our application description for Water-Kernel-NS (see Section 7.2.1)[13]. If the breakup penalty is to remain constant, MGS overhead must also parallelize to the same degree. This implies that the total volume of coherence actions performed by MGS must remain relatively the same when machine size is increased. Unfortunately, poor data locality causes total MGS traffic to increase as machine size is increased. Therefore, the breakup penalty goes up.

A large multigrain potential is observed throughout, though it reduces slightly as machine size is scaled. At 128 processors, the multigrain potential is 191% (down slightly from 197% for 32 processors), and at 512 processors, it further reduces to 174%. The implication is that providing hardware support for shared memory within SSMP nodes is useful, even for large machines. Furthermore, we observe that most of the multigrain potential is captured at modest SSMP node sizes in Figures 7.11 and 7.12. For a 128-processor machine, 84% of the multigrain potential is achieved by an SSMP node size of 8 processors, and for a 512-processor machine, 72% of the multigrain potential is achieved by an SSMP node size of 16 processors. This implies that large machines can be built using reasonably-sized SSMPs and still capture most of the benefit of having hardware-supported shared memory within SSMP nodes (we will comment more on SSMP node size in large-scale machines in the discussion on machine size and SSMP node size tradeoffs

[12]This comparison is rough because the data on page 134 is for Water-Kernel which uses a smaller CUTOFF constant and thus performs less work than Water-Kernel-NS.

[13]While Water does achieve good speedup, the assumption that parallel runtime reduces linearly with machine size is at best optimistic, and the parallel runtimes we use for the hardware DSM data points are necessarily too low. Therefore, the prediction of the model may be overly pessimistic with regards to breakup penalty.

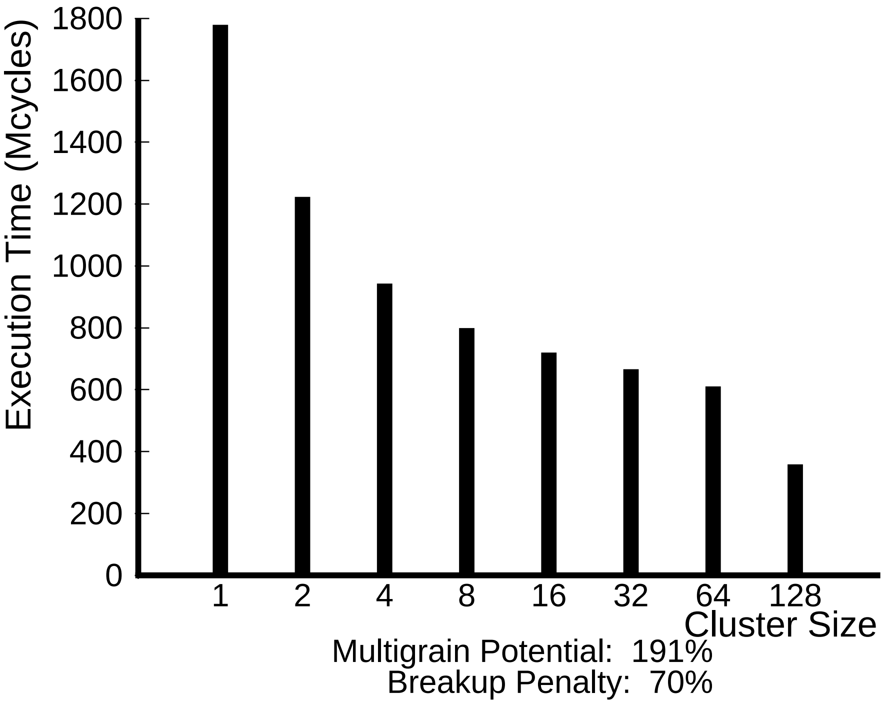


Figure 7.11: Scaling Water-Kernel-NS. 128 Processors.

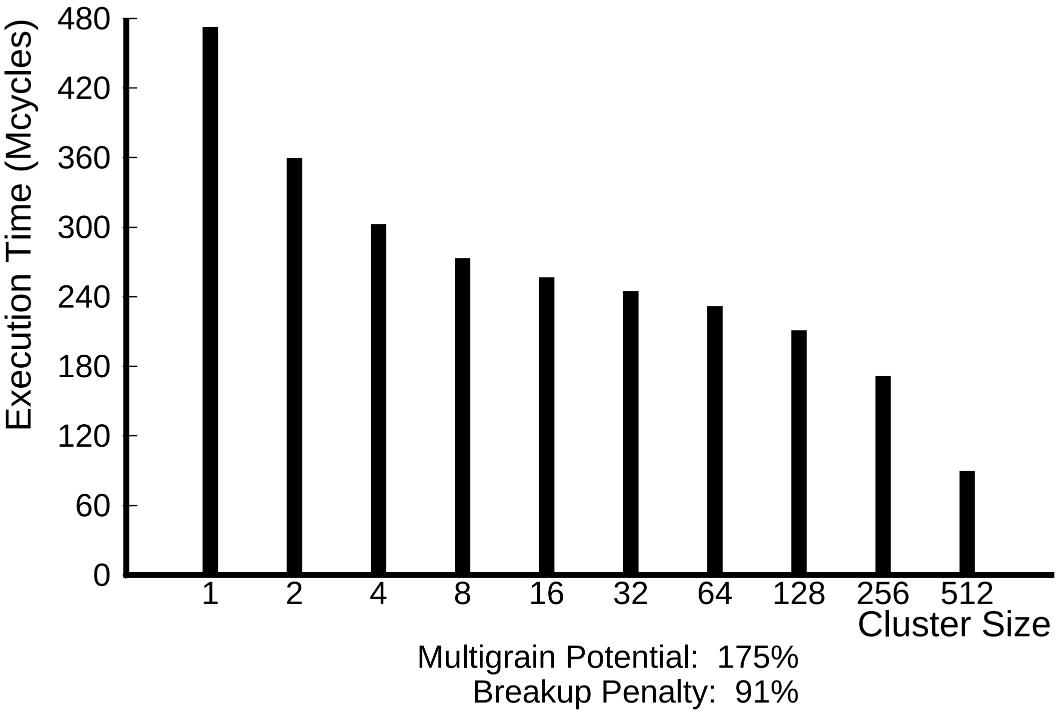


Figure 7.12: Scaling Water-Kernel-NS. 512 Processors.

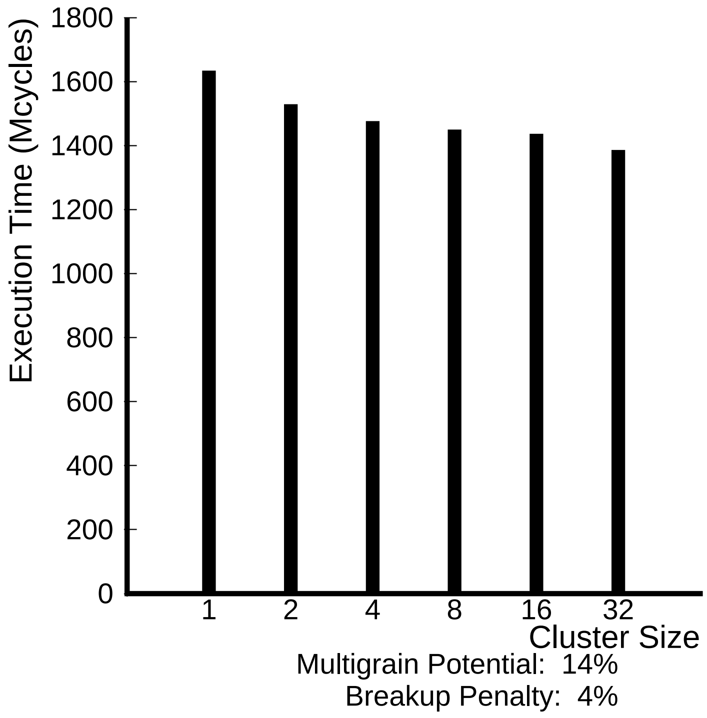


Figure 7.13: Scaling tiled Water-Kernel-NS. 32 Processors.

below).

The combination of large multigrain potentials with large breakup penalties results in modest overall performance. The speedup numbers for Water-Kernel-NS reported in Table 7.5 show that the large breakup penalties prevent DSSMPs from achieving linear speedup on the Water-Kernel-NS code. However, performance still scales such that significant speedups can be obtained on large DSSMPs. For instance, a DSSMP configuration with 512 total processors, built using 32 SSMPs, each with 16 processors achieves a speedup of approximately 130. Our conclusion for Water-Kernel-NS is that though its scalability is limited by the large breakup penalties, significant performance can nevertheless be achieved on large DSSMPs.

Figures 7.13, 7.14, and 7.15 show the scaling results for 32, 128, and 512 processors, respectively, on the tiled version of Water-Kernel-NS. As before, we use a larger problem size of 2048 molecules.

Again, we can see the impact of scaling problem size from 512 molecules to 2048 molecules by comparing Figure 7.13 to the right set of bars in Figure 6.13 on page 134. The main difference is the multigrain potential is much smaller at the larger problem size–it drops from 120% to only 14%. In the tiled version of Water-Kernel-NS, we do observe sharing granularity becoming coarser when problem size is increased. While the original version of Water-Kernel-NS does not exhibit this effect because of poor data locality, the tiled version does since data locality is significantly improved by the tiling transformation. As a result, it becomes less important to support fine-grain sharing mechanisms thus

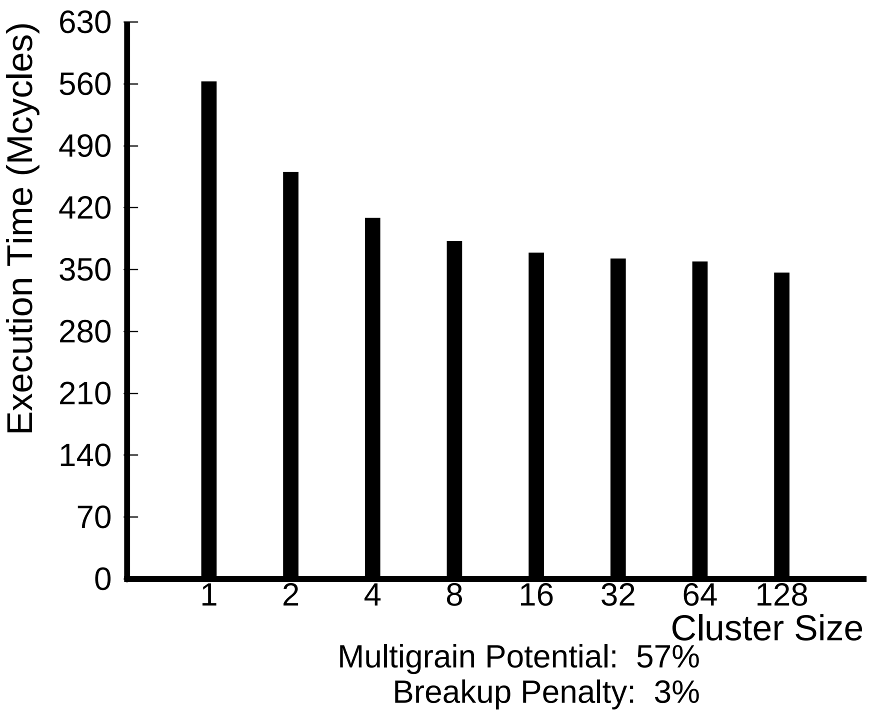


Figure 7.14: Scaling tiled Water-Kernel-NS. 128 Processors.

reducing the advantage of providing hardware-supported shared memory within SSMP nodes. The breakup penalty in Figure 7.13 remains small after scaling problem size as expected (4% as compared to 24% in Figure 6.13).

Figures 7.14 and 7.15 show scaling results on 128 and 512 processors. First, we observe that the breakup penalty remains almost imperceptible (3% for both 128 and 512 processors). Second, the multigrain potential increases as machine size grows–57% for 128 processors (up from only 14% for 32 processors), and 223% for 512 processors. Increasing the machine size while keeping the problem size fixed reduces sharing granularity. The tiling transformation tries to keep sharing between SSMPs coarse by containing fine-grain sharing within SSMP nodes (see the description of the tiling transformation in Section 6.4.1 of Chapter 6); however, when SSMP node size is small, the transformation is unable to successfully contain communication within SSMP nodes resulting in frequent inter-SSMP communication.

As in the original version of Water-Kernel-NS, most of the multigrain potential is captured at reasonable SSMP node sizes: 88% of the multigrain potential is achieved by an SSMP node size of 8 processors on a 128-processor DSSMP, and 94% of the multigrain potential is achieved by an SSMP node size of 16 processors on a 512-processor DSSMP. Again, as in the original version of Water-Kernel-NS, this result implies that DSSMPs can be built from small-scale multiprocessors.

Overall, the scaling results on the tiled version of Water-Kernel-NS are extremely encouraging. As the speedup numbers in Table 7.5 indicate, the tiled version of Water-

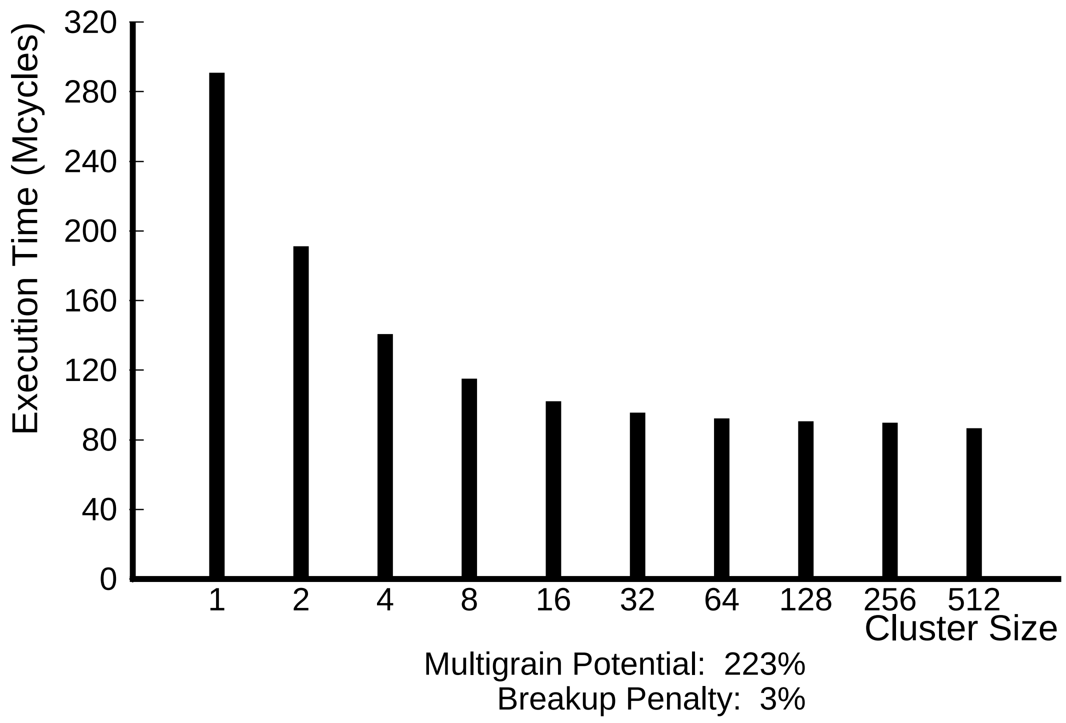


Figure 7.15: Scaling tiled Water-Kernel-NS. 512 Processors.

Kernel-NS achieves very good speedups for small and large machines. At the larger machines, the speedups are significantly better when SSMP node size is 8 processors or higher, thus indicating that DSSMPs hold a significant performance advantage over all-software DSMs. The conclusion for the tiled version of Water-Kernel-NS is that it exhibits excellent scalability on DSSMPs.

Machine Size and SSMP Node Size Tradeoff

The presentation of the scaling results in the above discussion demonstrates the impact of scaling both machine size and SSMP node size, but only in an orthogonal fashion. In this section, we examine the relationship between machine size and SSMP node size simultaneously. The goal of our evaluation is to expose the tradeoff between machine size and SSMP node size in building large-scale DSSMPs.

The speedup results reported in Table 7.5 indicate two performance trends: performance increases both as machine size is increased for a given SSMP node size, and as SSMP node size is increased for a given machine size. This suggests that DSSMP architects can achieve a desired level of performance by scaling machine resources along both the machine size and SSMP node size dimensions, either independently, or simultaneously. To provide a direct visualization of this two-dimensional scaling, and thereby enabling the visualization of the tradeoff between machine size and SSMP node size, we present our model results in a fashion that shows the impact of both machine size and

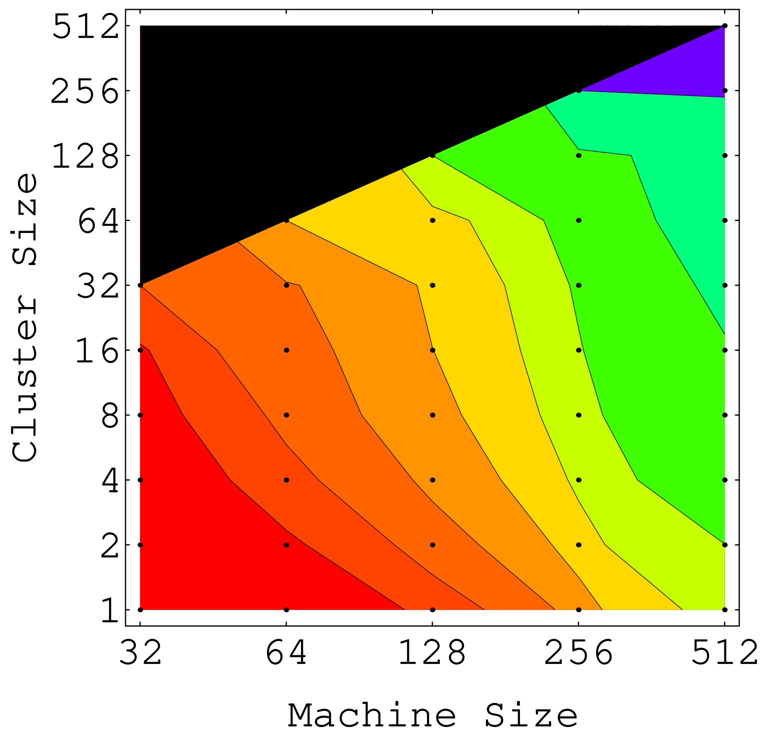


Figure 7.16: Performance-equivalent machines for the original version of Water-Kernel-NS.

SSMP node size on performance simultaneously.

Figure 7.16 shows a plot of performance-equivalent machines for the original version of Water-Kernel-NS. The X-axis of Figure 7.16 specifies the different machine sizes analyzed using our model, while the Y-axis specifies the different SSMP node sizes at each machine size. The plot is bounded from above by a line which intersects all machines composed from SSMP nodes that are equal in size to the total machine size, *i.e.* these are the all-hardware shared memory machines. Machines above this line are undefined since it is impossible to have an SSMP node size that is larger than the total machine size. The plot is bounded from below by the X-axis which intersects all machines built from uniprocessor nodes, *i.e.* these are the all-software shared memory machines. All the machines in the space between the upper and lower bounds represent DSSMP architectures. In addition, points have been placed in the plot to indicate those machines in the machine space for which performance has been predicted by our model. Finally, contours have been drawn through this space of machines to indicate those machines that deliver equivalent performance on the original version of Water-Kernel-NS. The spacing between contours has been chosen such that every 2nd contour represents a factor of two in performance, increasing from the origin to the upper-right corner of the plot.

By tracing the machines intersected by each contour, it is possible to identify different shared memory architectures that achieve the same level of performance. For instance, the plot shows that a 128-processor all-hardware DSM is approximately equivalent in

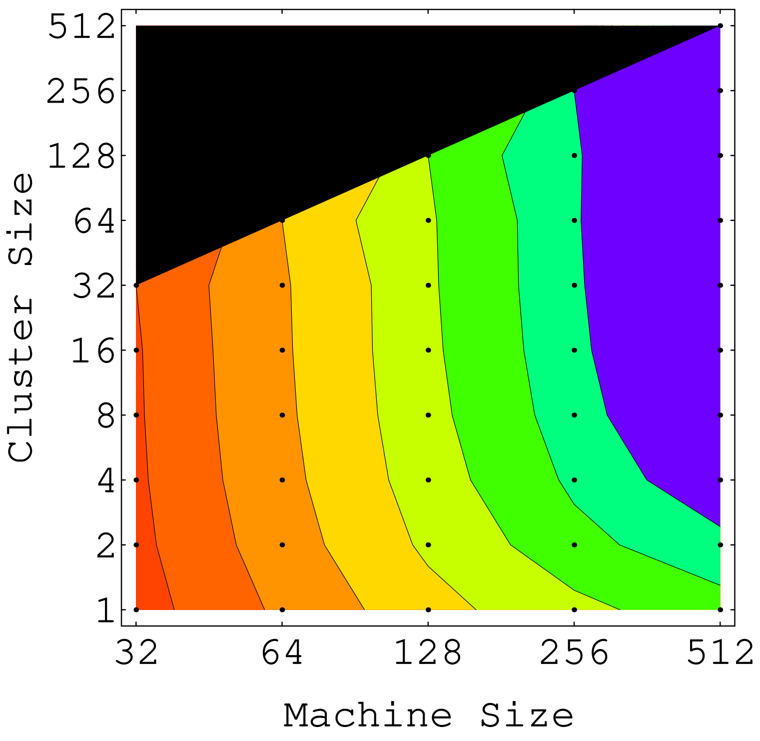


Figure 7.17: Performance-equivalent machines for the tiled version of Water-Kernel-NS.

performance to a 256-processor DSSMP built from SSMPs that are of size 16 processors each. These two machines are also equivalent in performance to a 512-processor DSSMP built from SSMPs that have an SSMP node size of 2 processors each. Similar comparisons can be made between architectures along other contours, *i.e.* at different levels of performance.

The contours in Figure 7.16 are diagonal, sloping from the upper-left corner to the lower-right corner of the plot. The slope of the contours, which will vary from application to application, determines the relationship between machine size and SSMP node size for that application. A steeper slope indicates that performance at a given machine size is less sensitive to SSMP node size. Applications with steep contours permit DSSMPs built from small-scale SSMPs to be competitive with all-hardware shared memory systems of the same size (in total processors). Applications with less steep contours require DSSMPs built from small-scale SSMPs to have a larger total processor count as compared against an all-hardware DSM of the same performance level. The slope of the contours for Water-Kernel-NS indicate that a factor of two increase in machine size is roughly equivalent to a factor of eight reduction in SSMP node size.

Figure 7.17 shows the performance-equivalent machines for the tiled version of Water-Kernel-NS. The effect of the tiling transformation, which improves the data locality exhibited by Water-Kernel-NS, is to steepen the slope of the contours. This reflects the result we have already seen: the loop tiling transformation makes DSSMPs built from small-scale SSMPs competitive in absolute performance with all-hardware DSMs of the same size. The contours in Figure 7.17 indicate that DSSMPs with SSMP node sizes as small as 8 or 16 processors closely match the performance of all-hardware DSMs at every machine size studied. Note, however, that below an SSMP node size of 8, especially for the larger machine sizes, the slope of the contours become less steep thus indicating that the scalability of performance tapers off at the smallest SSMP nodes. We conclude that even for applications with good locality, the scalability of all-software shared memory systems does not match that provided by DSSMPs built with small-scale SSMPs.

Chapter 8

Related Work

This chapter discusses related work in three parts. First, we discuss the impact of traditional software distributed shared memory systems on the MGS system in Section 8.1. Second, we discuss in Section 8.2 systems that, like MGS, employ multiple granularities for the coherence unit. Finally, we discuss other distributed shared memory systems that leverage SMPs as DSM nodes.

8.1 Page-Based Shared Memory

The initial idea to implement shared memory using a shared virtual address space thus enabling the construction of DSMs using commodity nodes originated from Kai Li's Ph.D. work [48]. Since then, several page-based software DSM systems have been proposed, many of which have been discussed and cited in Chapter 2.

MGS heavily leverages the body of work on software DSMs since MGS uses the same mechanisms proposed for traditional software DSM systems to provide shared memory across SSMPs. The system that has had the most impact on the design of MGS by far is Munin [13]. MGS borrows directly from Munin the use of delayed coherence to minimize communication using the Delayed Update Queue structure, and the implementation of multiple writers to reduce false sharing via twinning and diffing (see Section 2.2.2 of Chapter 2 for details on these mechanisms).

MGS borrows the delayed coherence and multiple-writer mechanisms from Munin because they provide good performance and because their implementation is relatively straight forward. Better software DSM performance is possible using a slightly more complex implementation of release consistency known as Lazy Release Consistency (LRC) [37] which has been implemented in the Treadmarks system [36]. LRC reduces the number of inter-node messages by further delaying when coherence happens. In Munin (and thus MGS), coherence is enforced *eagerly* at every release point where data is produced. Enforcing coherence at the producer is pessimistic and may result in unnecessary communication. LRC delays coherence until the acquire point where data is consumed. By waiting until the acquire point, the software DSM layer can provide a coherent view of data only to those processors which have the potential to use the data.

An implementation of MGS that uses LRC rather than the mechanisms from Munin would achieve higher performance. However, we do not expect such an implementation to make a qualitative change to the conclusions of this thesis. Regardless of the implementation of software DSM, it remains that software mechanisms are much more expensive than hardware mechanisms. Therefore, the major conclusions of this thesis hold. For instance, in an LRC-based implementation of MGS, it would still be necessary to have a Single-Writer-like mechanism that would remove all software coherence management on pages that exhibit single-writer sharing in order to deliver hardware levels of performance. Also, we still expect the difficult applications in Chapter 6 to exhibit the same qualitative results on an LRC-based MGS (*i.e.* large multigrain potentials and large breakup penalties), and we still expect the transformations identified for those applications to have the same order-of-magnitude performance impacts because they eliminate most of the software DSM overheads.

8.2 Multigrain Systems

In this section, we discuss the shared memory systems that have proposed using multiple granularities to support coherence. Of all the related work covered in this chapter, these systems have the most in common with MGS.

Coupling hardware cache-coherent shared memory with software page-based shared memory was first suggested in [16]. Their work investigates the performance of a system with up to 64 processors built using 8-way SMPs connected across an ATM network. Software shared memory between SMPs is provided using the LRC protocol. The evaluation is simulation based in which a very simple machine model was employed. The simulation treats all the processors in the same SMP as a single DSM node. Therefore, none of the design nor performance issues associated with integrating hardware and software shared memory were explored.

The system with the greatest similarity to MGS is SoftFLASH from Stanford [22]. SoftFLASH is a multigrain shared memory system implemented on a cluster of SGI Challenge SMPs connected across a switched HIPPI high-speed LAN. SoftFLASH implements a page-based version of the FLASH multiprocessor [44] coherence protocol; the page-based software DSM layer is integrated into the Irix 6.2 Unix kernel.

Several differences distinguish the SoftFLASH work from our work. First, unlike MGS which uses an experimental platform to evaluate multigrain shared memory, SoftFLASH is built on a commercial system and thus explores many interesting issues associated with the implementation of multigrain shared memory on top of an industry-grade operating system. One finding is that TLB consistency is expensive due to the high cost of synchronizing multiple processors through the Irix kernel. For SoftFLASH, a TLB consistency operation takes approximately 350 μsec on a 12-way SMP. On MGS, TLB consistency is much faster because the use of software virtual memory allows invalidation of mapping state through shared memory operations[1]. In general, such efficient management of mapping state is possible only if address translation is performed in software (see discussion on Shasta below).

Another finding in SoftFLASH is that limited inter-node network bandwidth can negatively impact performance. In SoftFLASH, the available bandwidth between processors in each SMP and the external network is fixed even when the number of processors in the SMP is scaled. In MGS, as the SSMP node size is scaled, the available inter-SSMP communication bandwidth increases as the square root of the SSMP node size, an artifact of our virtual clustering methodology implemented on top of Alewife's two-dimensional mesh topology. MGS also places a lower demand on inter-SSMP communication bandwidth since it uses a smaller page size, 1 K-bytes compared to the 16 K-byte pages used in SoftFLASH. The observations on inter-node bandwidth made in SoftFLASH point to the importance of providing scalable communications interfaces for DSSMPs. [34] proposes a scalable inter-SSMP communication interface for MGS that uses standard Internet protocols, and studies the effects of contention in the communication processors that run the communication protocol stacks.

Finally, the effects of false sharing are greater in SoftFLASH than in MGS for two reasons. First, SoftFLASH implements a single-writer protocol whereas MGS supports multiple writers via twinning and diffing. We have found that supporting multiple writers is important to reduce inter-SSMP communication. Second, as was stated above, SoftFLASH uses larger pages than MGS. As Section 6.5.2 of Chapter 6 showed, large page sizes can have a negative impact on performance for those applications that are vulnerable to false sharing.

MGS is also very similar in architecture to the system studied in [50]. In this work, a 16-processor system configured using 4-way SMPs over an ATM network is simulated. The authors study the effectiveness of prefetching techniques to hide the large latencies associated with paging between SMP nodes. They propose a prefetching technique tailored for software DSM systems, called "history prefetching," which uses dynamic access and synchronization information to predict future accesses. Therefore, this approach deals with the high cost of software page-based shared memory by trying to *tolerate* latency. In contrast, MGS leverages the hardware shared memory mechanisms within each SSMP node as much as possible to *reduce* latency. Successful containment of communication within SSMP nodes also allows MGS to support the difficult communication requirements of fine-grain applications.

In addition to the mixed hardware and software shared memory systems described thus far, there have been all-software systems that support multiple coherence granularities as well. Two examples of such software-only multigrain systems are CRL [35] and Shasta [57]. We first describe CRL, then Shasta.

CRL is a software-only implementation of shared memory that exports a *regions* pro-

[1] Even if we were to implement TLB consistency using interrupts, the implementation would still be far more efficient due to the support for fast interrupts on Alewife.

gramming interface[2]. A region is a programmer-defined block of memory that the shared memory layer uses to enforce coherence[3]. Multi-granular sharing occurs since the programmer is free to define different region sizes on a per data structure basis. Along with defining regions, the programmer also delimits accesses performed on region data. The shared memory layer uses these application-level annotations to bundle synchronization along with data delivery. CRL provides two advantages over conventional all-software systems. First, because the coherence unit can be tailored to the access patterns of an application, CRL eliminates false sharing communication. Second, the bundling of synchronization along with data prevents inefficient ping-pong sharing patterns that arise when there is simultaneous true sharing. CRL enjoys these advantages because the programmer or compiler explicitly provides granularity and synchronization information to the shared memory layer through program annotations. MGS is distinct from CRL in that it works on unmodified programs.

Since CRL was implemented on the same experimental platform[4], Alewife, and since many of the same benchmarks that were used in this thesis were studied on CRL, a comparison between CRL and MGS performance is meaningful in that it represents a true "apples-to-apples" comparison. Overall, CRL performance is impressive. For the Water and Barnes-Hut applications, the software-only CRL system was able to come within 15% and 12%, respectively, of Alewife performance on a 32-node machine. For the same applications on MGS (in their unmodified form), performance does not compare nearly as favorably. MGS is 159% and 191% worse on Water and Barnes-Hut, respectively, than native Alewife performance. However, with the application transformations described in Section 6.4 in Chapter 6, the comparison is much closer. The discrepancies between MGS and Alewife performance drop to 24.5% and 12.6%, respectively. CRL achieves slightly better performance in the case of Water, but performance is matched in the case of Barnes-Hut.

While the two systems are quite similar in the performance they deliver, a comparison of the CRL and MGS design philosophies uncover significant differences. The primary issue on which the two systems differ is how to treat the programming model. In CRL, the programming model is designed with performance in mind since the programmer deals with both correctness and performance simultaneously when developing applications using the regions abstraction. This encourages programming disciplines that lead to high performance. MGS decouples programming from performance since programmers on the MGS system can choose to ignore performance when developing correct programs. However, to achieve CRL-like performance on MGS, the programmer must observe the locality issues that were addressed in Chapter 6. One benefit of a decoupled approach

[2]In addition to CRL, another system that supports the regions abstraction is described in [56].

[3]The use of application-specific data layout information by the shared memory layer is similar to what is supported under Entry Consistency as implemented by the Midway distributed shared memory system [7].

[4]The Alewife implementation of CRL only uses the efficient communication interfaces provided by Alewife. Alewife's hardware support for shared memory was purposefully bypassed to measure the discrepancy in performance when shared memory mechanisms are provided in software only.

is that for applications where performance doesn't matter, it may be feasible to ignore locality thus requiring less effort (unmodified shared memory programmers will run correctly on MGS). For those applications where performance does matter, programmer or compiler effort can be focused on portions of the application that are performance critical. Considering the relative difficulty in programming for a regions abstraction versus performing program transformations like the ones identified in Section 6.4 is interesting, but one that is difficult to evaluate due to its subjectivity.

Finally, while both CRL and MGS are concerned about scalability, MGS in addition addresses commoditization. A CRL system requires tight coupling between all its nodes since providing efficient communications interfaces monolithically is crucial for performance. MGS tries to tolerate loose coupling by using tight coupling in a local fashion thus permitting commodity interfaces between groups of tightly-coupled processors.

Shasta is another software-only shared memory system that supports multi-granular sharing; however, its approach is quite different from CRL. Instead of embedding the notion of granularity in the programming model as is done in CRL, Shasta tries to automatically detect the natural sharing granularity in an application through the compiler, and then convey this granularity information to a software shared memory layer so that a customized coherence unit can be used on a per data structure basis. A variable coherence unit is enabled by employing software address translation and protection checking and allowing the compiler to control the size of each memory mapping[5]. Shasta proposes several compiler optimizations that reduce the cost of software translation and checking code so that fine-grain access control can be supported efficiently. In [57], the authors report translation and checking overheads in the range of 5% – 35% (compare to 50% – 100% for MGS, with one application exceeding 100%).

Like Shasta, MGS supports software address translation as well, but for a different purpose–to remedy the lack of hardware support for virtual memory in Alewife. However, Shasta suggests that software virtual memory is a feasible approach, even in a production system. Using software instrumentation for translation and checking in a production DSSMP is attractive, and can solve two practical problems. First, it can allow a smaller coherence unit (even a variable-sized coherence unit as is done in Shasta) between SSMPs on platforms that cannot support small pages. The importance of using smaller pages to reduce false sharing was demonstrated in Section 6.5.2 of Chapter 6. Second, the software instrumentation code can be extended to also perform TLB consistency. This can address the high cost of TLB consistency mechanisms in production operating systems (see SoftFLASH discussion above). For instance, Shasta performs TLB invalidation by polling in software for TLB invalidation events, thus removing the need to synchronize processors through the kernel each time TLB consistency is required.

[5]The general approach of using software address translation and checking in Shasta resembles the Blizzard-S work [59].

8.3 Fine-Grained SMP Clusters

Building scalable shared memory systems using SMPs has become an area of increasing interest in the most recent years due to the commoditization of the SMP architecture. SoftFLASH and MGS are examples of such systems that support both a fine-grain (cache-line) and coarse-grain (page) coherence unit. In contrast, several systems recently proposed have also adopted SMPs as DSM building blocks, but support fine-grain transfers between SMP nodes. Such *fine-grained SMP clusters* have the potential to offer higher performance than multigrain systems like SoftFLASH and MGS since they offer more efficient inter-SMP communication mechanisms, but they require some additional special-purpose hardware to support the fine-grain transfers.

One approach to building fine-grained SMP clusters is to build hardware support for fine-grain transfers in the network interface (NI) to the SMP. The DEC Memory Channel [26] and the SHRIMP network interface [10] are examples of such "intelligent" NIs. These network interfaces support fine-grain communication between workstations by providing a remote write mechanism between host memories. Special transmit and receive regions can be defined to the network interface in each host's address space. Writes performed to transmit regions are sent through the network interface and appear in the corresponding receive regions on remote workstations, all in hardware. While the NI hardware supports transfers, it doesn't maintain coherence between the regions. Therefore, a coherence protocol, in software, is still necessary to enforce coherence, but the protocol can leverage the remote write mechanism as an efficient fine-grain messaging facility. Examples of SMP clusters that use fine-grain NI-based communication are [40] and [8]. [40] describes a 32-node cluster of 4-way DEC AlphaServer SMPs connected by the Memory Channel network. They implement two software coherence protocols, the Cashmere [62] protocol and the Treadmarks [36] protocol, and compare their performance. [8] describes a 16-node cluster of 4-way PC-based SMPs connected by the SHRIMP network interface. Coherence is provided by the AURC [30] protocol.

A fine-grained SMP cluster with even less hardware than the intelligent NI approach described above is the Wisconsin COW [58]. This system is comprised of 40 dual-processor Sun SPARCStation 20s connected across a Myrinet network [11]. The COW uses less hardware than the NI-based approaches because transfers between SMPs (in addition to coherence) are off-loaded onto one of the processors on the host SMP. Fine-grain transfers are enabled by a small piece of checking hardware, called the Typhoon-0 board. This checking hardware sits on the memory bus of each SMP, snooping for transactions that cause access violations. Tags are maintained in the Typhoon-0 hardware to allow access control at cache-line granularity. Once a violation is detected, the checking hardware traps one of the host processors to service the violation in software. The coherence protocol handlers are implemented at user level; therefore, applications can link against a library of common protocols, or provide a protocol that is tailored to the application for higher performance [53].

Fine-grained SMP clusters offer an interesting alternative to multigrain systems. Like multigrain systems, they represent an intermediate architecture between traditional all-software and all-hardware DSMs. Because they leverage some additional special-purpose hardware, they are slightly more costly than DSSMPs, but the use of such minimal hardware allows them to support communication between SMP nodes more efficiently. However, even with hardware support, communication between processors on separate SMPs will remain more costly than communication between colocated processors, so we expect many of the same locality issues addressed in this thesis will apply to fine-grained SMP clusters.

Chapter 9

Conclusion

This thesis addresses the tension between cost and performance in the design of scalable shared memory multiprocessors. The crux of the thesis lies in the observation that traditional shared memory systems, *i.e.* fine-grain hardware cache-coherent and coarse-grain software page-based architectures, cannot effectively address both cost and performance because of their monolithic construction.

Hardware cache-coherent distributed shared memory architectures are designed to deliver scalable high performance across a wide range of applications. However, the efficient shared memory interfaces they provide come at the cost of tight coupling between all processing elements. As system size is scaled for higher performance, such tight coupling becomes difficult to maintain in a cost-efficient manner. On the other hand, software page-based shared memory architectures abandon fine-grain support in favor of less costly coarse-grain mechanisms. Because coarse-grain mechanisms don't support the same aggressive interfaces as fine-grain mechanisms, they do not require special-purpose hardware. Instead, these systems can leverage commodity communications interfaces and support shared memory purely in software, leading to highly cost-effective designs. Unfortunately, the lack of efficient shared memory interfaces means that coarse-grain architectures cannot support fine-grain applications.

This thesis responds to the high cost of hardware cache-coherent architectures and the low performance of software page-based architectures by proposing a new way to construct large-scale shared memory multiprocessors: couple multiple small- to medium-scale parallel workstations using page-based software shared memory techniques. Our approach synthesizes a single transparent shared memory layer through the cooperation of both fine-grain and coarse-grain shared memory mechanisms.

This *multigrain approach* to building systems strives to meet the goals of both scalable performance and cost-effective design. By leveraging the small- to medium-scale shared memory multiprocessor, or SSMP, large-scale systems inherit the performance and cost benefits offered by SSMPs. SSMPs are equipped with hardware shared memory interfaces that can efficiently support fine-grain sharing. However, because the SSMP is designed for smaller-scale configurations, the fine-grain interfaces do not present the same engineering challenges of providing tight coupling across a large-scale machine. Furthermore, SSMPs are commodity systems because there is a high-volume demand for smaller-scale multiprocessors in the server market. The commodity status of SSMPs makes them extremely cost-effective components.

The effectiveness of multigrain architectures has been thoroughly evaluated in this thesis. An in-depth study of several applications along with a detailed mathematical analysis of system behavior has produced a body of scientific evidence characterizing the performance of these systems. In the rest of this chapter, we present several conclusions regarding the effectiveness of multigrain architectures based on this experimental and analytic evidence. Our conclusions can be divided into four categories: application performance, 1.1–1.3, program optimization, 2.1–2.3, SSMP node size, 3, and analytic techniques, 4.

> Conclusion 1.1: *Applications that exhibit coarse-grain sharing are insensitive to the underlying implementation of shared memory.*

Our experimental study revealed four applications that we categorize as "Easy" applications. All of these applications exhibit coarse-grain sharing patterns. We find that these applications achieve good performance regardless of the underlying mechanisms for shared memory provided by the system. In terms of our performance framework, we observe a flat performance profile as SSMP node size is varied, resulting in a very small multigrain potential and breakup penalty. While it is assuring that these applications perform well, they are uninteresting from a systems evaluation standpoint given that their system needs can be so easily met.

> Conclusion 1.2: *Applications that exhibit fine-grain sharing benefit from the fine-grain mechanisms supported by multigrain architectures. As a result, DSSMPs provide significantly higher performance on these fine-grain applications as compared to conventional software DSM architectures.*

Multigrain architectures are effective at supporting some of the fine-grain sharing exhibited by applications with more demanding communications requirements, *i.e.* applications in what we call the "Challenging" and "Pathologic" categories. The four applications from our experimental study that exhibit fine-grain sharing all demonstrate significant performance improvements on multigrain architectures as compared to all-software architectures. The experimental evidence shows these applications perform up to 61% to 88% faster (*i.e.* the multigrain potential ranges from 61% to 88%) when they are provided some hardware-supported shared memory. This empirical evidence is corroborated by our analytic evaluation which predicts for one of the fine-grain applications that the multigrain potential is still significant, over 170%, as total machine size is scaled to 512 processors. The strong evidence showing a performance advantage on multigrain architectures as compared to all-software systems allows us to conclude that SSMPs are by far better building blocks than uniprocessor workstations for large-scale multiprocessors.

> Conclusion 1.3: *Fine-grain sharing patterns that extend across the entire machine cannot be supported efficiently by the software mechanisms between*

SSMPs in multigrain architectures. On applications that exhibit such global fine-grain sharing, all-hardware architectures hold a significant performance advantage over multigrain architectures.

Unfortunately, the comparison of multigrain architectures with all-hardware systems is less favorable than with all-software systems on the fine-grain applications in our study. Our experimental study shows that hardware cache-coherent machines perform 159% to 1014% faster (*i.e.* the breakup penalty ranges from 159% to 1014%) than multigrain architectures on these difficult applications. Our analysis corroborates the performance advantage for all-hardware systems at large machine sizes, predicting a 70% and 91% breakup penalty on the one application we analyzed for 128- and 512-processor systems. While the performance advantage of multigrain architectures over all-software architectures suggests that some fine-grain sharing is captured by efficient mechanisms within SSMP nodes, the performance discrepancy between multigrain architectures and all-hardware systems indicates that these applications have significant amounts of fine-grain sharing that span the entire machine. Since multigrain systems only provide fine-grain support locally within SSMPs, this global fine-grain sharing cannot be handled efficiently.

Besides lower performance relative to all-hardware systems, another consequence of the large breakup penalties is that absolute speedup achieved on multigrain systems for these difficult applications is significantly lower than what can be achieved on all-hardware systems. We observe almost no speedup for the "Pathologic" applications, and only modest speedups for the "Challenging" applications (up to 10.4 for Water and up to 4.6 for Barnes-Hut on a 32-processor DSSMP). The poor performance observed for the applications in the "Pathologic" category leads us to conclude that these applications, in their unmodified form, cannot be supported on multigrain systems (see discussion below on application transformations). For the applications in the "Challenging" category, we conclude that multigrain architectures can deliver acceptable levels of performance.

The results of the analytic study strongly suggest that the modest levels of performance achieved on "Challenging" applications allow multigrain architectures to be competitive in cost-performance to all-hardware systems. Our analysis indicates that large multigrain architectures can provide significant speedups on the Water-Kernel-NS workload, even though they are not linear. For instance, the model predicts that 16 16-way SSMPs achieve a speedup of approximately 92. To match this performance level, an all-hardware system requires 128 processors. Whether the multigrain system is more favorable than the all-hardware system of equivalent performance depends on the relative costs of the two architectures. Overall, our analysis shows that for Water-Kernel-NS, an increase in total machine size by a factor of 2 allows a decrease in SSMP node size by a factor of 8. Architects of large-scale shared memory machines can combine such equivalent performance information with cost information for a target technology. Doing so allows an architect to trade off machine size and SSMP node size to position an architecture at a desirable cost-performance point.

Conclusion 2.1: *Global fine-grain sharing can be localized through program transformations that increase data locality. Such transformations yield signif-*

icant performance improvements on multigrain architectures. Most notably, they allow multigrain systems to be competitive in absolute performance with all-hardware systems.

As we observed above, applications that exhibit fine-grain sharing tend to do so in a global fashion. In these instances, we find that fine-grain sharing can be confined within SSMP nodes through program transformations that enhance data locality. Such transformations allow applications with demanding communications requirements to better leverage the fine-grain shared memory mechanisms provided by multigrain systems. The data locality transformations have a dramatic impact on performance for applications in both the "Challenging" and "Pathologic" categories. In fact, the improvements in performance are so dramatic that they allow multigrain architectures to become competitive with all-hardware systems in terms of absolute performance. When the transformations are applied, both of the "Challenging" applications and one of the "Pathologic" applications exhibit breakup penalties inside 40%. TSP, the other "Pathologic" application, has a moderate breakup penalty of 66%.

Conclusion 2.2: *Even with the data locality transformations, it is still important to provide fine-grain shared memory mechanisms.*

Even after the data locality transformations have been applied, multigrain architectures still hold a significant performance advantage over all-software systems. Applications in the "Challenging" category maintain similar multigrain potentials as before the transformations were applied (58% for Water, and 81% for Barnes-Hut), and applications in the "Pathologic" category exhibit enormous multigrain potentials (282% for TSP and 812% for Unstructured). This result leads to the interesting conclusion that there is something fundamental about the nature of fine-grain sharing in these applications. We find that while transformations can significantly reduce the amount of fine-grain sharing, the transformations do not eliminate fine-grain sharing; instead, they limit the extent to which fine-grain sharing occurs across the machine so that it can be contained within SSMP nodes. Therefore, supporting such applications efficiently requires fine-grain shared memory mechanisms even when programming or compiler effort is applied to increase locality.

We recognize that it is almost always possible to improve the performance of applications by expending large amounts of programming effort. Any conclusions drawn based on the transformation studies must also consider each transformation's implementation effort. This leads to our next conclusion.

Conclusion 2.3: *Most of the transformations encountered in our study are simple. We believe they are within the capabilities of current-day optimizing compilers or moderately-skilled programmers.*

Of the four applications studied in Section 6.4.2 of Chapter 6, we consider three of the applications to require transformations that have been commonly performed for shared memory machines. Water-Kernel received a loop tiling transformation that is similar to loop tiling already performed by existing optimizing compilers. Barnes-Hut and TSP require transformations that address hot-spotting on a small number of shared data structures. While we do not know of existing automatic techniques for these transformations, the code modifications are simple. Arguably, they can be viewed as "fixes" to poor parallel programming discipline in the original codes, and in fact, one of the transformations we perform appears almost exactly in a later release of the Barnes-Hut code by its authors (which in fact, we discovered retroactively). In addition to the contention-relieving transformation, Barnes-Hut also received another transformation that addresses false sharing. The false sharing transformation we perform resembles a transformation that exists in a current compiler. The last application, Unstructured-Kernel, is by far the most difficult to tune because the sharing patterns are dynamic, irregular, and highly data-dependent. While our transformations for Unstructured-Kernel achieve extremely good performance, we do not consider them within the bounds of "reasonable programming effort." Certainly, they cannot be implemented using compilation techniques.

Overall, the conclusions regarding our program optimization experiences elude to a tradeoff between system cost and performance resilience. One view of multigrain architectures is that they have an equivalent performance *potential* as all-hardware cache-coherent shared memory systems on difficult fine-grain applications. When fine-grain sharing in difficult applications is clustered, the multigrain architecture can match the all-hardware architecture in absolute performance. The same cannot be said about all-software architectures which lack the fine-grain mechanisms necessary to efficiently support fine-grain applications even when applications exhibit locality. However, the potential for MPP-like performance on multigrain systems is realized only with some extra programming or compiler effort. All-hardware systems don't require this extra effort because they are more resilient to applications that exhibit poor locality. In return for a lower resilience to poor locality, multigrain architectures provide scalable cost, an advantage they hold over all-hardware architectures.

> Conclusion 3: *Most of the multigrain potential can be achieved by small SSMP nodes, for example 8 or 16 processors, implying that effective multigrain systems can be constructed from small-scale multiprocessor nodes.*

The multigrain approach espouses building large-scale systems by building tightly coupled mechanisms in hardware to the extent that cost remains reasonable, then resorting to software mechanisms to provide scaling beyond that point in a cost-effective manner. While this provides an upper bound on SSMP node size constrained by cost, deciding on the actual SSMP node size also requires considering a lower bound, constrained by performance.

For most of the applications in the "Challenging" and "Pathologic" categories (in which clustering actually matters), much of the multigrain potential is achieved at SSMP node sizes of 4 and 8. For the original versions of Water, TSP, and Unstructured, between 60% and 70% of the multigrain potential is achieved by an SSMP node size of 4, and between 80% and 90% by an SSMP node size of 8. Barnes-Hut requires larger SSMP node sizes as only 59% of the multigrain potential is achieved by an SSMP node size of 8. Even greater performance is observed at small SSMP nodes for the transformed versions of the applications. Water-Kernel, TSP, and Unstructured-Kernel achieve between 80% and 90% of the multigrain potential by an SSMP node size of 4, and between 90% and 100% by an SSMP node size of 8. Again, Barnes-Hut requires larger SSMP nodes as only 60% of the multigrain potential is achieved by an SSMP node size of 8.

The results provided by the experimental study are encouraging, however, they only represent performance on a small machine, 32 processors. Our analytic study provides insight into the impact of SSMP node size on larger systems. Similar results were found for the one application we studied, Water-Kernel-NS. For the original version of Water-Kernel-NS, 84% of the multigrain potential is achieved at an SSMP node size of 8 processors on a 128-processor DSSMP, and 72% at an SSMP node size of 16 on a 512-processor DSSMP. The results for the tiled version of Water-Kernel-NS are even more encouraging (88% and 94%, respectively). Since our analysis only considers one application, further study is needed on more applications to provide conclusive results for large machines; nevertheless, initial indications are that small SSMP nodes are adequate for building high-performance DSSMPs.

The last conclusion relates to our experiences with analysis on multigrain architectures.

> Conclusion 4: *Synchronization analysis can accurately predict performance on software shared memory systems.*

The analytic work presented in this thesis provides important insight into how large DSSMPs will perform, and how SSMP node size scales with increasing total machine size. These byproducts of the analysis work have been discussed throughout this conclusion chapter. An equally important contribution of the analysis work, and one with implications that reach beyond the results of this thesis, is the general notion of using synchronization analysis to reason about the performance of software shared memory systems.

The agreement between our model predictions and measured results suggests that synchronization analysis can accurately predict performance on multigrain systems. Granted, our analytic study only looks at a single application; more evidence is necessary before we can be completely confident that the approach is robust. Nevertheless, the experience we have with synchronization analysis in this thesis is promising. In addition to showing agreement on more applications, the overall success of synchronization analysis also depends on the ability to generalize the approach. The idea of synchronization analysis itself is general; however, there were many instances where for expediency, we special-cased the analysis for the application at hand. For instance, the interleave analysis presented in Section 7.2.1 of Chapter 7 is applicable only to the Water code. It is not difficult to imagine a more general analysis that could apply to any loop nest.

We believe that with further research, synchronization analysis can become an important tool for both multigrain systems and software shared memory systems in general.

Synchronization analysis can be used as a methodology for studying scalability, as was done in this thesis. It can also be integrated into optimizing compilers to evaluate the efficacy of different locality transformations. Finally, it could be used by a performance analysis tool that gives feedback to programmers of multigrain systems so that performance bottlenecks can be identified quickly.

Bibliography

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

[2] Anant Agarwal, Johnathan Babb, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Gino Maa, Ken Mackenzie, Dan Nussbaum, Mike Parkin, and Donald Yeung. Sparcle: Today's Micro for Tomorrow's Multiprocessor. In *HOTCHIPS*, August 1992.

[3] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, Honolulu, HI, June 1988. IEEE.

[4] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of SIGPLAN '93, Conference on Programming Languages Design and Implementation*, June 1993.

[5] Rajeev Barua, David Kranz, and Anant Agarwal. Addressing Partitioned Arrays in Distributed Memory Multiprocessors - the Software Virtual Memory Approach. In *Proceedings of the Fifth MIT Student Workshop on Scalable Computing*, Wellesley, MA, July 1995.

[6] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. CMU-CS 91-170, Carnegie Mellon University, September 1991.

[7] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference*, pages 528–537, February 1993.

[8] Angelos Bilas, Liviu Iftode, David Martin, and Jaswinder Pal Singh. Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance. Technical Report TR-517-96, Princeton University, 1996.

[9] David Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, Boston, MA, April 1989. ACM.

[10] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. TR 437-93, Princeton University, November 1993.

[11] Nanette J. Boden, danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[12] Ralph Butler and Ewing Lusk. User's Guide to the p4 Programming System. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

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

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

[15] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[16] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, Chicago, IL, April 1994.

[17] Alan L. Cox and Robert J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. Technical Report 263, University of Rochester Computer Science Department, May 1989.

[18] William J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[19] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[20] S. J. Eggers and R. H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, New York, June 1989. IEEE.

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

[22] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 210–221, Cambridge, Massachusetts, October 1996. ACM.

[23] Anant Agarwal et. al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[24] Charles M. Flaig. Vlsi mesh routing systems. Master's thesis, California Institute of Technology, Department of Computer Science, California Institute of Technology, 256-80, Pasadena, CA 91125, May 1987.

[25] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[26] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2), February 1996.

[27] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[28] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, 1989.

[29] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. *Computer Architecture News*, pages 39–50, 1993.

[30] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[31] B. L. Jacob and T. N. Mudge. Software-Manged Address Translation. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, San Antonio, TX, February 1997. IEEE.

[32] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, pages 74–77, June 1990.

[33] Tor E. Jeremiassen and Susan J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, July 1995.

[34] Xiaohu Daniel Jiang. A scalable parallel inter-cluster communication system for clustered multiprocessors. Master's thesis, Massachusetts Institute of Technology, Department of Computer Science and Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139, August 1997.

[35] Kirk Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995.

[36] Pete Keleher, Alan Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proceedings of the 1994 Usenix Conference*, pages 115–131, January 1994.

[37] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[38] Kendall Square Research, Inc., 170 Tracer Lane, Waltham, MA 02154. Kendall Square Research Technical Summary, 1992.

[39] Leonard Kleinrock. *Queueing Systems*, volume I. John Wiley & Sons, New York, 1975.

[40] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, MichałCierniak, Srinivasan Parthasarathy, Wagner Meira, Jr., Sandhya Dwarkadas, and Michael Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 157–169, Denver, CO, June 1997.

[41] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, New York, May 1993. ACM.

[42] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Supercomputing Conference*, Tokyo, Japan, July 1993.

[43] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. In *Proceedings International Symposium on Shared Memory Multiprocessing*, Japan, April 1991. IPS Press.

[44] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994. IEEE.

[45] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[46] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[47] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

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

[49] Beng-Hong Lim. Parallel C Functions for the Alewife System. Alewife Memo 37, MIT Laboratory for Computer Science, August 1993.

[50] Magnus Karlsson and Per Stenström. Performance Evaluation of a Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*. IEEE, February 1996.

[51] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[52] Shubu Mukherjee, Shamik Sharma, Mark Hill, Jim Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Proceedings of the 5th Annual Symposium on Principles and Practice of Parallel Programming*, pages 68–79. ACM, July 1995.

[53] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, New York, April 1994. IEEE.

[54] Bryan S. Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. *ACM Operating Systems Review*, 23(5):137–146, December 1989.

[55] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[56] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Principles and Practices of Parallel Programming, 1993*, pages 229–238, San Diego, CA, May 1993.

[57] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, October 1996. ACM.

[58] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, Jarmes R. Larus, Christopher E. Lukas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnar, and David A. Wood. Implementing Fine-Grain Distributed Shared Memory on Commodity SMP Workstations. Technical Report TR-1307, University of Wisconsin-Madison, March 1996.

[59] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[60] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. Load Balancing and Data Locality in Hierarchical N-body Methods. Technical Report CSL-TR-92-505, Stanford University, 1992.

[61] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[62] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 19th Annual Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[63] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[64] Patricia J. Teller and Marc Snir. TLB Consistency on Highly Parallel Shared-Memory Multiprocessors. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 184–193, 1988.

[65] The Ultra Enterprise 1 and 2 Server Architecture. SUN Microsystems. Mountain View, CA, 1996.

[66] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995.

[67] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, May 1992.

[68] W.-D. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*. IEEE, June 1989.

[69] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 1991.

[70] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, 1989.

[71] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[72] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 1996 International Symposium on Computer Architecture*, Philadelphia, May 1996.

Appendix A

MGS Protocol Specification

This Appendix provides a complete, while at the same time compact, specification of the MGS multigrain shared memory protocol. The original specification for the protocol appears in [72], an early paper describing the MGS work. The specification provided in this appendix represents an updated version of that original MGS system.

The specification consists of three parts. First, the state transition diagrams for the three state machines, Local Client, Remote Client, and Server, described in Section 4.2 of Chapter 4, are presented in Figures A.1, A.2, and A.3, respectively. Second, the state transition tables for each machine, which annotate the state transition diagrams, appear in Tables A.1, A.2, and A.3, respectively. Finally, Table A.4 lists all the message types used in the MGS system.

The state transition tables provide the precondition and postconditions for each transition in the state transition diagrams. Each table consists of a set of rows where each row refers to one or more transitions in the corresponding state transition diagram. Each row is divided into six columns. The column labeled "Arc" provides the identification number which relates the table entry to a specific transition in one of the three state transition diagrams. The "Event" column shows the event or incoming message type that triggers the state transition. There are three different types of triggering events: "RTLBFault," "WTLBFault," and "Release." The first two events are TLB faults due to read and write accesses, respectively. The third event is a release operation emitted by the application. All other entries in the "Event" column are incoming messages.

Next, the "Precondition" column specifies all conditions which must hold true in order for the transition to occur. The column labeled "L" is part of the precondition and indicates the action taken on the page lock corresponding to the page involved in the state transition. A "+" signifies that the lock must be acquired before the precondition is satisfied; otherwise, a "-" appears indicating that no lock acquire is necessary. A second value specifies the state of the lock after the state transition completes. The lock is either relinquished or held, denoted by "R" and "H," respectively. Notice that there are no entries in the "L" column for the Server machine since page locks are for synchronizing clients only. Finally, the last two columns of the state transition tables specify the consequences of each state transition. The "Side Effects" column indicates changes to various protocol data structures, and the "Out Message" column specifies all the outgoing messages sent after the transition is completed as well as their destinations.

A few words should be mentioned on the notation used throughout several of the entries in the state transition tables. Italicized identifiers represent sets of processor IDs. For instance, *tlb_dir*, *read_dir*, and *write_dir* are all sets whos members are processors being tracked by the respective directories. The notation $< message > \Rightarrow < pid >$ denotes that we send $< message >$ to $< pid >$, while the notation $< message > \Rightarrow < set >$ denotes that we send $< message >$ to every processor specified in $< set >$. $|< set >|$ denotes the number of elements in $< set >$. $< set > -> tail$ returns $< set >$ minus the first element. "l_home" and "g_home" denote the ID of the processor that owns the local physical copy and the home copy of a page, respectively. "pagestate" refers to the access privilege, and "mapping" refers to the page mapping, for the local physical copy of the page in question. "src" refers to the processor ID of the sender of the current incoming message.

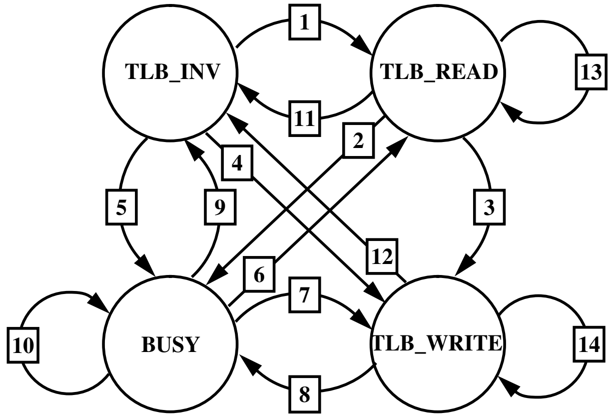


Figure A.1: MGS Protocol state transition diagram: Local-Client Machine.

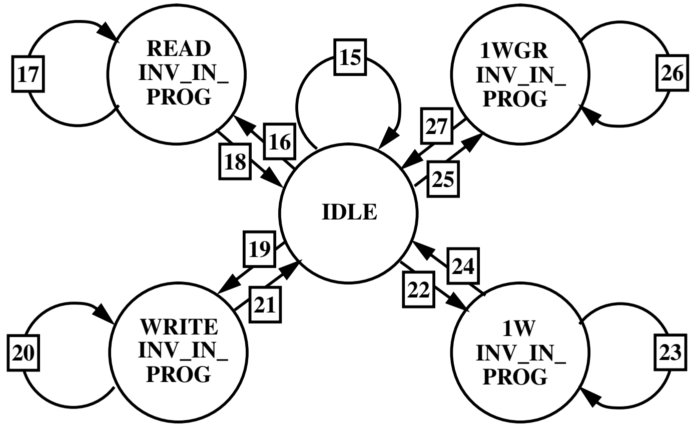


Figure A.2: MGS Protocol state transition diagram: Remote-Client Machine.

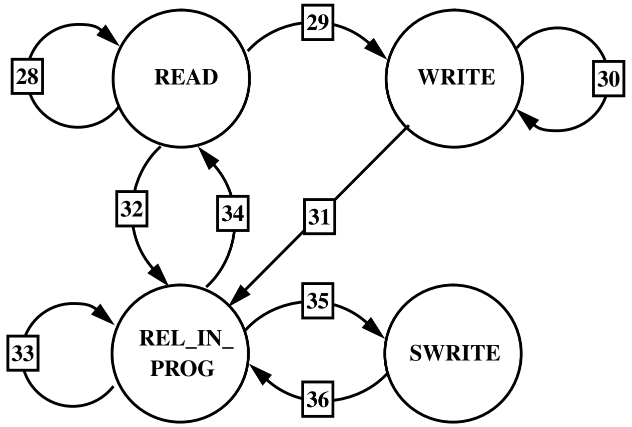


Figure A.3: MGS Protocol state transition diagram: Server Machine.

Figure A.1: MGS Protocol state transition diagram: Local-Client Machine
Figure A.2: MGS Protocol state transition diagram: Remote-Client Machine
Figure A.3: MGS Protocol state transition diagram: Server Machine

Arc	Event	Precondition	L	Side Effects	Out Message
1	RTLBFault	pagestate != INV	+/R	mapping $\rightarrow$ TLB, $tlb\_dir = tlb\_dir \cup \{\text{src}\}$	
2,5	WTLBFault	pagestate == READ	+/H	mapping $\rightarrow$ TLB, $tlb\_dir = tlb\_dir \cup \{\text{src}\}$	UPGRADE $\Rightarrow$ l_home
3,4	WTLBFault	pagestate == WRITE	+/R	mapping $\rightarrow$ TLB, $tlb\_dir = tlb\_dir \cup \{\text{src}\}$ $DUQ = DUQ \cup \{\text{addr}\}$	
5	RTLBFault	pagestate == INV	+/H		RREQ $\Rightarrow$ g_home
	WTLBFault	pagestate == INV	+/H		WREQ $\Rightarrow$ g_home
6	RDAT		–/R	map page, $tlb\_dir = \{\text{src}\}$, pagestate = READ	
7	WDAT		–/R	map page, $tlb\_dir = \{\text{src}\}$, pagestate = WRITE $DUQ = DUQ \cup \{\text{addr}\}$	
	UP_ACK		–/R	$DUQ = DUQ \cup \{\text{addr}\}$	
8	Release		+/H	addr = $DUQ\text{–>}\ head$, $DUQ = DUQ\text{–>}\ tail$	REL $\Rightarrow$ g_home(addr)
9	RACK	$DUQ == \phi$	–/R		
10	RACK	$DUQ\ != \phi$		addr = $DUQ\text{–>}\ head$, $DUQ = DUQ\text{–>}\ tail$	REL $\Rightarrow$ g_home(addr)
11	PINV			invalidate TLB	PINV_ACK $\Rightarrow$ l_home
12	PINV			invalidate TLB, $DUQ = DUQ - \{\text{addr}\}$	PINV_ACK $\Rightarrow$ l_home
13,14	P2INV			DUQ = DUQ – {addr}	P2INV_ACK $\Rightarrow$ l_home

Table A.1: MGS Protocol state transition table: Local-Client Machine.

Arc	Event	Precondition	L	Side Effects	Out Message
15	UPGRADE			make twin, pagestate = WRITE	UP_ACK $\Rightarrow$ src, WNOTIFY $\Rightarrow$ g_home
16	INV	pagestate == READ	+/H	clean page, free page, count = $\mid tlb\_dir \mid$	PINV $\Rightarrow$ tlb_dir
19	INV	pagestate == WRITE	+/H	make diff, free page, count = $\mid tlb\_dir \mid$	PINV $\Rightarrow$ tlb_dir
22	1WINV		+/H	clean page, count = $\mid tlb\_dir \mid$	PINV $\Rightarrow$ tlb_dir
25	1WGR		+/H	count = $\mid tlb\_dir \mid$	P2INV $\Rightarrow$ tlb_dir
17,20,23	PINV_ACK	count != 0		count = count – 1	
26	P2INV_ACK	count != 0		count = count – 1	
18	PINV_ACK	count == 0	–/R	$tlb\_dir = \phi$, pagestate = INV	ACK $\Rightarrow$ g_home
21	PINV_ACK	count == 0	–/R	$tlb\_dir = \phi$, pagestate = INV	DIFF $\Rightarrow$ g_home
24	PINV_ACK	count == 0	–/R	$tlb\_dir = \phi$	1WDATA $\Rightarrow$ g_home
27	P2INV_ACK	count == 0	–/R		ACK1W $\Rightarrow$ g_home

Table A.2: MGS Protocol state transition table: Remote-Client Machine.

Table A.1: MGS Protocol state transition table: Local-Client Machine.
Table A.2: MGS Protocol state transition table: Remote-Client Machine.

Table A.3: MGS Protocol state transition table: Server Machine.

Arc	Event	Precondition	L	Side Effects	Out Message
28,30	RREQ			$read\_dir = read\_dir \cup \{\text{src}\}$	RDAT $\Rightarrow$ src
29,30	WREQ			$write\_dir = write\_dir \cup \{\text{src}\}$	WDAT $\Rightarrow$ src
29	WNOTIFY			$read\_dir = read\_dir - \{\text{src}\}$, $write\_dir = write\_dir \cup \{\text{src}\}$	
31	REL	$\mid write\_dir \mid\ != 1$		count $= \mid read\_dir \cup write\_dir \mid$, $rl = \{\text{src}\}$, $rd = wr = \phi$	INV $\Rightarrow$ $read\_dir \cup write\_dir$
	REL	$\mid write\_dir \mid\ == 1$, $\mid read\_dir \mid\ != 0$		count $= \mid read\_dir \cup write\_dir \mid$, $rl = \{\text{src}\}$, $rd = wr = \phi$	INV $\Rightarrow$ $read\_dir$, 1WINV $\Rightarrow$ $write\_dir$
	REL	$\mid write\_dir \mid\ == 1$, $\mid read\_dir \mid\ == 0$		count = 1, $rl = \{\text{src}\}$, $rd = wr = \phi$	1WGR $\Rightarrow$ $write\_dir$
32	REL			count $= \mid read\_dir \cup write\_dir \mid$, $rl = \{\text{src}\}$, $rd = wr = \phi$	INV $\Rightarrow$ $read\_dir$
33	ACK	count != 0		count = count − 1	
	DIFF	count != 0		count = count − 1, buffer diff data	
	1WDATA	count != 0		count = count − 1, copy data to home	
	RREQ			$rd = rd \cup \{\text{src}\}$	
	WREQ			$wr = wr \cup \{\text{src}\}$	
	REL			$rl = rl \cup \{\text{src}\}$	
	WNOTIFY				
	ACK1W	$\mid rd \cup wr \mid\ != 0$		count = 1	INV $\Rightarrow$ $write\_dir$
34	ACK	count == 0		merge diffs, $read\_dir = write\_dir = \phi$	RACK $\Rightarrow$ rl, RDAT $\Rightarrow$ rd, WDAT $\Rightarrow$ wr
	DIFF	count == 0		merge diffs, $read\_dir = write\_dir = \phi$	RACK $\Rightarrow$ rl, RDAT $\Rightarrow$ rd, WDAT $\Rightarrow$ wr
	1WDATA	count == 0		$read\_dir = write\_dir = \phi$	RACK $\Rightarrow$ rl, RDAT $\Rightarrow$ rd, WDAT $\Rightarrow$ wr
35	ACK1W	$\mid rd \cup wr \mid\ == 0$			RACK $\Rightarrow$ rl
36	RREQ			count = 1, $rd = \{\text{src}\}$, $rl = wr = \phi$	INV $\Rightarrow$ $write\_dir$
	WREQ			count = 1, $wr = \{\text{src}\}$, $rl = rd = \phi$	INV $\Rightarrow$ $write\_dir$

Table A.3: MGS Protocol state transition table: Server Machine.

Local Client ⇒ Remote Client Messages	
UPGRADE	Upgrade Local Page from Read to Write Privilege
PINV_ACK	Acknowledge TLB Invalidation
P2INV_ACK	Acknowledge DUQ Invalidation
Remote Client ⇒ Local Client Messages	
UP_ACK	Acknowledge Upgrade
PINV	Invalidate TLB Entry
P2INV	Invalidate DUQ Entry
Local Client ⇒ Server Messages	
RREQ	Read Data Request
WREQ	Write Data Request
REL	Release Request
Server ⇒ Local Client Messages	
RDAT	Read Data
WDAT	Write Data
RACK	Acknowledge Release
Remote Client ⇒ Server Messages	
ACK	Acknowledge Read Invalidate
DIFF	Acknowledge Write Invalidate and Return Diff
1WDATA	Acknowledge Single Writer Invalidate and Return Data
WNOTIFY	Notify Upgrade from Read to Write Privilege
ACK1W	Acknowledge Single Writer Status
Server ⇒ Remote Client Messages	
INV	Invalidate Page
1WINV	Invalidate Single-Writer Page
1WGR	Grant Single-Writer Status

Table A.4: Message types used to communicate between the Local-Client, Remote-Client, and Server machines in the MGS Protocol.