Multigrain Shared Memory

[†]Donald Yeung, [‡]John Kubiatowicz, and [§]Anant Agarwal[*]
[†]University of Maryland at College Park
[‡]University of California at Berkeley
[§]Massachussetts Institute of Technology

October 6, 2000

Abstract

Parallel workstations, each comprising tens of processors based on shared memory, promise cost-effective scalable multiprocessing. This paper 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. We call these systems Distributed Shared-memory MultiProcessors (DSMPs).

This paper introduces the design of a shared memory system that uses multiple granularities of sharing, called MGS, and presents a prototype implementation of MGS on the MIT Alewife multiprocessor. Multigrain shared memory enables the collaboration of hardware and software shared memory, thus synthesizing a single transparent shared memory address space across a cluster of multiprocessors. The system leverages the efficient support for fine-grain cache-line sharing within multiprocessor nodes as often as possible, and resorts to coarse-grain page-level sharing across nodes only when absolutely necessary.

Using our prototype implementation of MGS, an in-depth study of several shared memory applications is conducted to understand the behavior of DSMPs. Our study is the first to comprehensively explore the DSMP design space, and to compare the performance of DSMPs against all-software and all-hardware DSMs on a single experimental platform. Keeping the total number of processors fixed, we show that applications execute up to 85% faster on a DSMP as compared to an all-software DSM. We also show that all-hardware DSMs hold a significant performance advantage over DSMPs on challenging applications, between 159% and 1014%. However, program transformations to improve data locality for these applications allow DSMPs to almost match the performance of an all-hardware multiprocessor of the same size.

[*]This research was funded in part by DARPA contract N00014-94-1-0985 and in part by NSF Experimental Systems grant MIP-9504399. Copyright 2000 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

1 Introduction

Large-scale shared memory multiprocessors have received significant attention within the computer architecture community over the past decade, in large part due to two factors. First, the shared memory programming model is desirable because it relieves the programmer from the burden of explicitly orchestrating communication, as is required by a message passing communication model [1]. Second, large-scale shared memory machines have the potential for extremely good cost-performance. They are constructed using nodes that rely only on modest technology. Unlike traditional supercomputers, large-scale shared memory machines use commodity microprocessors, and achieve high performance by exploiting medium- to coarse-grain parallelism across a large number of nodes.

While the promise for cost-performance has contributed to the popularity of large-scale shared memory machines, this promise has thus far gone unfulfilled due to the high cost of providing efficient communication mechanisms at large scales. Traditionally, large-scale shared memory machines support efficient communication 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 between nodes through tightly coupled hardware interfaces. Although such efficient communication is crucial for scalable performance on communication-intensive applications, the investment in hardware mechanisms comes at a cost. 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. Perhaps most important, the large-scale nature of these machines prevents them from capitalizing on the economy of cost that higher volume small-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. 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 [2]. 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 on machines that have tightly-coupled special-purpose interfaces [3]. The high cost of communication on commodity systems prevents them from supporting applications with intensive communication requirements.

Traditional architectures for large-scale shared memory machines have not satisfactorily addressed the tension between providing efficient communication mechanisms for high performance and leveraging commodity components for low cost. In this paper, we explore a new approach to building large-scale shared memory machines that leverages small- to medium-scale shared memory multiprocessors as the building block for larger systems. Small-scale (2–16 processor) and medium-scale (17–128 processor) shared memory machines are commodity components. A familiar example is the bus-based symmetric multiprocessor. Another example is the small- to medium-scale distributed-memory multiprocessor. The latter architecturally resembles large-scale (greater than 128 processor) tightly-coupled machines, but is targeted for smaller systems. Since both symmetric multiprocessors and small- to medium-scale distributed-memory multiprocessors are relevant to our work, we will refer to both of them using the single term, *SMP*.[1]

The SMP is an attractive building block for large-scale multiprocessors for two reasons. First, SMPs 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 SMPs do not incur exorbitant costs because the tight coupling required is only provided across a small number of processors. Unlike large-scale hardware DSMs, SMPs can be cost-effective as evidenced by the commercial success of the bus-based symmetric multiprocessor.

We call a large-scale system built from a collection of SMPs a *Distributed Shared memory MultiProcessor* (DSMP). DSMPs are constructed by extending the hardware-supported shared memory in each SMP using software DSM techniques to form a single shared memory layer across multiple SMP 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 SMP. Page-based software DSM provides a larger page sharing grain between processors in separate SMPs.

Recently, several DSMP architectures have been constructed and studied [6, 7, 8, 9, 4]. This paper builds upon the work in [4] and makes the following novel contributions:

1. We present a fully functional design of a multigrain shared memory system, called *MGS*, and provide a prototype implementation of MGS on the Alewife multiprocessor.

2. We define two performance metrics, the *breakup penalty* and the *multigrain potential*, that characterize application performance on DSMPs.

3. We provide a performance evaluation of several shared memory programs on the MGS prototype. While other performance evaluations of DSMP systems have been conducted (see Section 6), our study is the first to explore the entire spectrum of DSMP architectures and to provide a consistent comparison of these architectures against traditional all-software and all-hardware DSMs on a single experimental platform.

4. We quantify the impact of program transformations for data locality to more effectively leverage the clustered nature of DSMPs.

[1]Traditionally, only symmetric multiprocessors have been called SMPs. In [4], which describes the original version of this work, we introduced the terminology *SSMP* (for *S*calable *S*hared memory *M*ulti*P*rocessor) as a general term to describe both bus-based and switched-interconnect architectures. However, in recent SMP machines, the trend has been to replace the shared bus with a switched interconnect [5] thus blurring the distinction between traditional SMPs and distributed-memory machines. Since manufacturers have called these new systems SMPs as well (thus making our original terminology a misnomer), we adopt the familiar SMP terminology for both bus-based and switched-interconnect architectures throughout the rest of this paper.

The rest of this paper is organized as follows. Section 2 describes the DSMP architecture and multigrain shared memory, and presents our performance metrics for DSMPs. Section 3 presents the MGS design, and section 4 discusses our prototype implementation of MGS on Alewife. Section 5 presents the experimental results, and Section 6 discusses related work. Finally, Section 7 presents our conclusions.

2 Multigrain Shared Memory

In this section, we describe a DSM architecture that supports shared memory in a multigrain fashion, called a DSMP. We also describe what we call DSMP families, and present a performance framework that allows us to reason about the performance of applications on DSMPs based on the notion of DSMP families.

2.1 DSMPs

Traditional all-hardware and all-software DSMs implement shared memory in a monolithic fashion. All-hardware systems support shared memory entirely in hardware using special-purpose interfaces leading to high-performance at the expense of high cost. All-software systems support shared memory entirely in software using commodity interfaces leading to low-cost at the expense of poor performance on communication-intensive applications. Due to their monolithic nature, these traditional architectures are positioned at two extremes across a wide spectrum of cost and performance. Unfortunately, the ability to trade off cost for performance along this spectrum does not currently exist for large-scale shared memory machines.

We propose an "intermediate architecture" between all-hardware and all-software DSMs, called **D**istributed **S**hared memory **M**ulti**P**rocessors (DSMPs). DSMPs provide *some* tight coupling, but not across the entire machine. "Neighborhoods" of tight coupling are formed using cache-coherent shared memory within small- to medium-scale multiprocessor nodes. 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*.

Figure 1 shows the major components in a DSMP. A DSMP is a distributed shared memory machine in which each DSM node is itself a multiprocessor. These nodes are small- to medium-scale cache-coherent shared memory machines. We envision that each node will either be a bus-based symmetric multiprocessor (such as the SGI Challenge), or a small- to medium-scale distributed-memory (NUMA) multiprocessor (such as the SGI Origin [10] in a small-scale configuration). Throughout this paper, we will refer to both types of node architectures using the same terminology, *SMP*.

As Figure 1 shows, DSMPs 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 SMP. In the case that the SMP is a symmetric multiprocessor, this network is a bus. In the case that the SMP is a NUMA multiprocessor, it may be a switched point-to-point network. The external network connects the individual SMPs and consists of a high-performance local area network (LAN), such as ATM or switched Ethernet.

In addition to a hierarchy of networks, DSMPs also provide shared memory support in a hier-

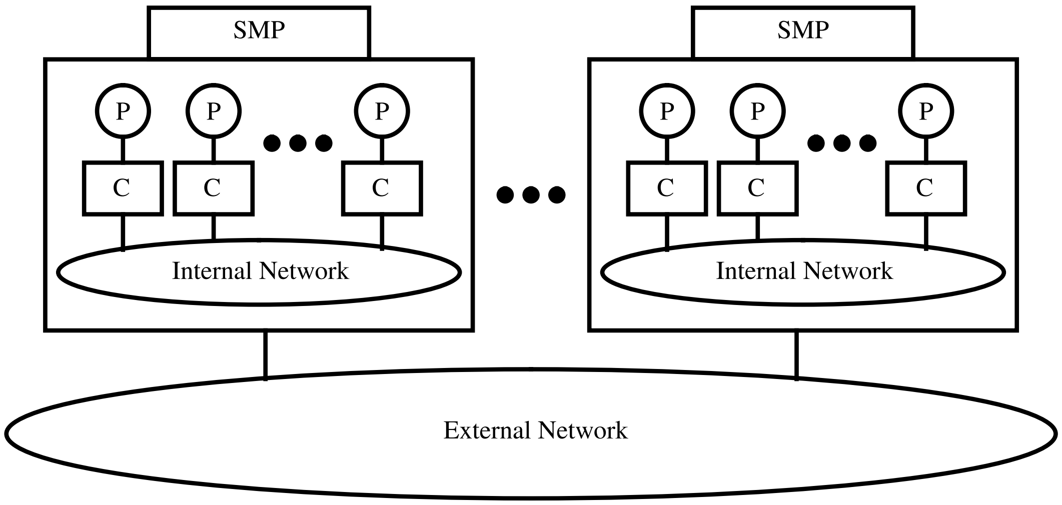

Figure 1: A Distributed Shared memory MultiProcessor (DSMP).

archical fashion. Each SMP provides special-purpose hardware for cache-coherent shared memory. This may take the form of snoopy-based cache coherence in the case of bus-based machines, or directory-based cache coherence in the case of NUMA multiprocessors. Between SMPs, shared memory is supported using page-based software shared memory.

DSMPs 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. In addition, 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.

2.2 DSMP Families

A key parameter that describes any parallel machine is the system size, or the number of processing elements in the system, P. DSMPs can also be characterized in this fashion; however, another key parameter in the case of DSMPs is the SMP node size, C. Therefore, the two parameters, P and C, identify specific DSMP configurations.

Many DSMP configurations are similar; in particular, we say that all configurations with the same P parameter belong to the same *DSMP family.* As illustrated in Figure 2, a family of DSMPs is defined by fixing the total number of processing elements, P, and varying SMP node size.[2] DSMPs in the same family differ only in the way processors are clustered. The clustering boundary, *i.e.* the boundary that divides processors on the same SMP from those that are on remote SMPS, determines where hardware-supported shared memory meets software-supported shared memory. Therefore, by varying SMP node size, we in effect vary the mix of fine-grain and coarse-grain support

[2]In this paper, we only consider SMP node sizes, C, that divide P evenly. Otherwise, the DSMP will contain SMPs of varying sizes, in which case a single SMP node size parameter cannot specify the sizes of all SMPs in the system.

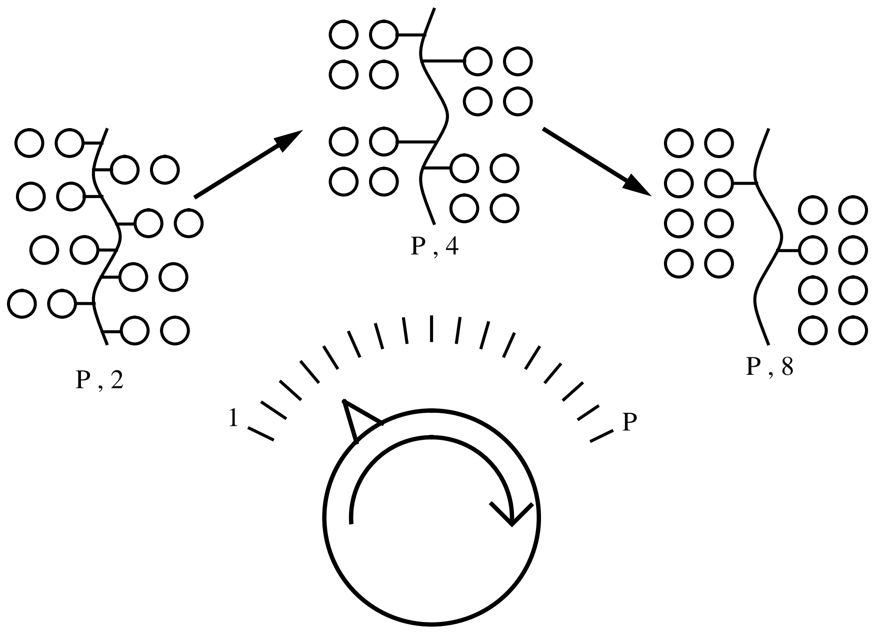

Figure 2: A family of DSMPs is defined by fixing the total processing and memory resources, and varying the SMP node size. The two parameters, P and C, denote a DSMP with P total processors, and C processors per SMP node.

for sharing between processors. DSMPs with smaller SMP nodes rely more on software-supported shared memory and provide more coarse-grain (page granularity) sharing support. Conversely, DSMPs with larger SMP nodes rely more on hardware-supported shared memory and provide more fine-grain (cache-line granularity) sharing support. Furthermore, monolithic shared memory machines are degenerate configurations: all-software DSMs have $C = 1$ (*e.g.* $P, 1$), and all-hardware DSMs have $C = P$ (*e.g.* P, P).

The most important aspect of the P, C parameters is that they point to the existence of a "knob," as depicted in Figure 2. This knob is not only an SMP node size knob and a sharing granularity knob, but it also serves as a knob for tuning cost against performance.

2.3 DSMP Performance Framework

A performance framework that characterizes application performance on DSMPs can be defined based on the notion of DSMP families and the node size knob. Given a DSMP with a fixed total machine size P, we measure an application's performance on the DSMP as the SMP node size C is varied from 1 to P. This set of measurements constitutes the application's *performance profile*. The performance profile tells us how an application responds to a change in the mixture of hardware and software in the implementation of shared memory. It reflects the degree of fine-grain hardware mechanisms needed relative to coarse-grain software mechanisms for the application to achieve a certain level of performance. Furthermore, since the endpoints of the performance profile, $C = 1$ and $C = P$, correspond to the all-software and all-hardware DSMs, respectively, the performance profile also compares DSMP performance to the performance achieved on monolithic (conventional) shared memory machines.

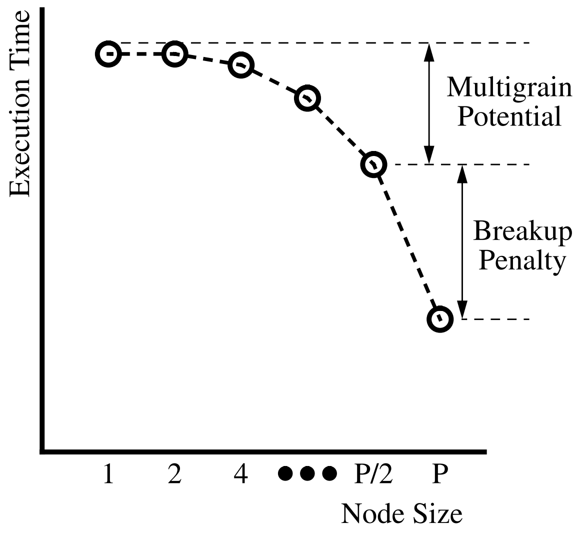

Figure 3: A hypothetical application analyzed using the performance framework. This application is not well-suited for DSMPs.

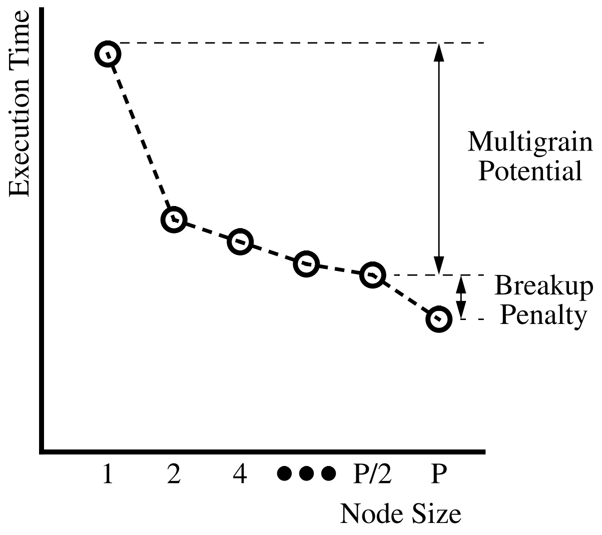

Figure 4: A hypothetical application analyzed using the performance framework. This application is well-suited for DSMPs.

Figure 3 shows the performance profile of a hypothetical application. Execution time is plotted against the SMP node size parameter, C, in powers of 2 for a total system size, P. We define two quantitative metrics that identify the most important features on the performance profile. These metrics have been labeled in Figure 3, and are:

Breakup Penalty. The execution time increase between the P SMP node size and the $\frac{P}{2}$ SMP node size is called the "breakup penalty." The breakup penalty measures 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 SMP node size of 1 and an SMP 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 SMP nodes.

Our performance framework tells us that the hypothetical application in Figure 3 is not well-suited for DSMPs. First, the application's performance profile has a large breakup penalty. This indicates that the application will perform poorly on the DSMP 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 SMP nodes; therefore, this application will not achieve much higher performance on a DSMP as compared to an all-software DSM.

In contrast, Figure 4 shows the analysis of another hypothetical application, again using our performance framework. The performance profile presented in Figure 4 displays a very small breakup penalty. This application will perform almost as well on a DSMP 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

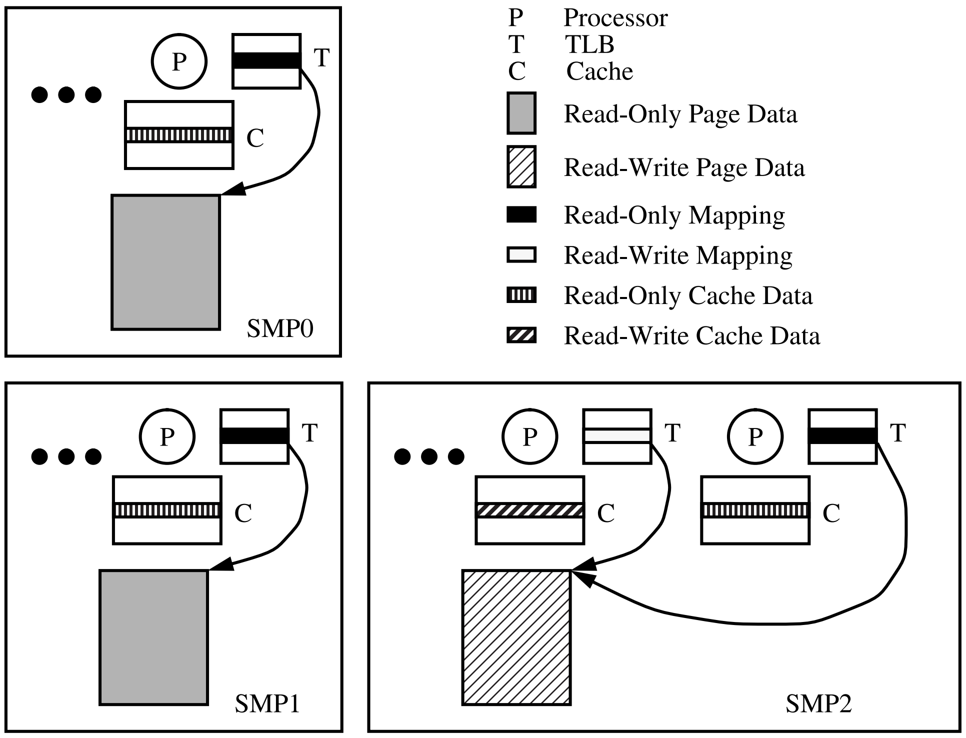

Figure 5: Distribution of a page of data across three SMPs in the MGS system. Only the processors involved in sharing are shown; there are other processors in each SMP that are not shown.

large benefits derived from capturing fine-grain sharing within SMP nodes. The implication for the application depicted in Figure 4 is that it will perform well on DSMPs.

Notice that while the breakup penalty and multigrain potential quantify an application's behavior on a DSMP, the values for the two metrics are specific to the machine on which they were measured. If a machine-independent characterization is desired, it may be necessary to acquire measurements for the breakup penalty and multigrain potential on multiple platforms.

3 MGS System Design

This section discusses the design of the MGS system, including several details about the protocol that implements multigrain shared memory. In Section 3.1, we give an overview by describing how data replication occurs in the system. Section 3.2 then discusses the software DSM protocol in greater detail, and Section 3.3 briefly describes the software state machines that implement the protocol. Our discussion defers platform-specific implementation issues of the system to Section 4.

3.1 System Overview

MGS supports replication of data at both page and cache-line granularities. Between SMPs, coherence actions occur at the granularity of a page. Once a page is resident in the memory of an SMP, processors within the SMP can map the page and further replicate the data at cache-line grain via hardware cache coherence.

Every virtual page in the MGS system has a unique *home* that contains the *physical home copy*. The location of the home is based on the virtual address and remains fixed for all time. SMPs other than the home which desire access to the page can replicate the page. Replicated pages can carry either read-only or read-write privilege. Once an SMP has a *physical local copy*, processors in the SMP can gain access to the data by first creating mappings for the page. Read pages can be mapped in read-only mode, while read-write pages can be mapped in either read-only or read-write mode. Once mapped, accesses can be made to the page, and replication of the data in the page occurs via hardware cache coherence.

Figure 5 illustrates how data from a single page gets distributed to SMPs and processors. SMP0 contains the physical home copy which is itself in read mode. A single processor in SMP0 has a read-only mapping and has read some of the data. SMP1 and SMP2 have read-only and read-write physical local copies, respectively. Since SMP1 has a read-only copy, its processors can only map it in read-only mode. Processors in SMP2, however, can map their physical local copy in either mode. One of the processors has a read-write mapping, while another has a read-only mapping.

3.2 MGS Software DSM

At the heart of the MGS system is the page-based software DSM protocol that provides shared memory between SMP nodes. The MGS software DSM protocol resembles Munin [11]: it is release consistent,[3] invalidation-based, and supports multiple writers. Like Munin, MGS uses the delayed update queue (DUQ) structure to track dirty pages and to propagate their changes back to the home location at release time. Also like Munin, MGS supports multiple writers by "twinning" all pages with read-write privilege, and computing diffs between the page and its twin at release time. Only portions of the page that have changed are propagated back to the home copy. Finally, the consistency in MGS is eager. At a release point, invalidations are performed immediately, and the home copy becomes consistent with respect to all processors and SMPs after the release completes. (An exception to eager consistency occurs for pages under the single-writer optimization, described in Section 3.2.3).

In addition to the basic Munin protocol, the MGS software DSM layer includes several extensions to adapt the protocol for SMP nodes: a per-node page table and TLB directory, separate per-processor DUQs, a single-writer optimization, and a page cleaning mechanism.

3.2.1 Page Table and TLB Directory

Each node in an MGS system maintains a page table and a TLB directory in software. The page table records the virtual-to-physical frame mapping and mapping privilege, either read-only or read-write, for every page resident in the node's memory (it does this both for pages that are home to the node and for pages that have been replicated from remote homes). For such resident pages, the TLB directory records the set of processors in the node that have mapped the page in their TLBs.

[3]We assume that hardware cache coherence on the SMPs presents a memory model that is release consistent, or that is stronger than release consistency. Since software shared memory between SMPs is release consistent, the overall model seen by the programmer is release consistency.

The page table is consulted each time a processor suffers a TLB fault, yielding three possible outcomes. If the desired page does not appear in the page table, then the page is not resident in the node and the software DSM layer is invoked to obtain a copy from the remote home. If an entry for the page is found but the page does not have sufficient privilege (*i.e.* a write fault occurs on a page with read-only privilege), then the DSM layer is invoked to request a privilege upgrade from the remote home. Finally, if an entry for the page is found and the privilege is sufficient, then the faulting processor performs a TLB fill from the page table, marks the TLB directory to reflect its copy of the mapping, and returns from the TLB fault.

The TLB directory is used to maintain coherence on mapping state cached in the processor TLBs. This is the well-known TLB coherence problem for which many solutions have been proposed [12, 13, 14, 15]. The solution used in MGS is closest to the one used in the PLATINUM system [12]. When the software DSM layer invalidates a page resident on a node, the TLB directory is consulted to determine the set of processors that have cached a mapping for the page. An invalidation request is posted to all processors in this copy set via inter-processor interrupts. Processors can be interrupted selectively because the TLB directory specifies the exact set of processors that require TLB invalidation. The page invalidation operation does not proceed until all interrupted processors have acknowledged that their TLB mappings have been invalidated.

In MGS, it is possible for concurrent accesses to occur on the page table and TLB directory entries for the same page. For instance, a request for invalidation from the software DSM layer may arrive at an SMP at the same time that a processor on the SMP tries to map the page. MGS provides a lock for every resident page to enforce atomic access. TLB fill and TLB invalidation operations must acquire ownership of the lock before modifying page table or TLB directory state. A single lock per resident page is sufficient since the associated page table and TLB directory entries are always modified together.

3.2.2 Per-Processor DUQs

The MGS system maintains a separate DUQ list for each processor in an SMP node. When a processor performs a release operation, only the pages modified by the processor performing the release as indicated by its private DUQ list are twinned, flushed back to their respective home copies, and invalidated. Separate DUQ lists prevent one processor's release from prematurely invalidating dirty pages modified by other processors in the same SMP. However, because multiple processors in the same SMP may be modifying the same page, invalidation operations must keep the separate DUQ lists coherent. In MGS, DUQ list coherence is piggy-backed onto the same interrupts that enforce TLB coherence.

3.2.3 Single-Writer Optimization

For higher performance, the MGS system tries to leverage the hardware-supported cache-coherent shared memory mechanisms provided within each SMP node as often as possible. While pages shared between processors on distinct nodes must invoke software, an opportunity exists to bypass the software DSM layer when sharing is confined to processors colocated on the same SMP. We call such pages *single-writer* pages because there is a single outstanding write copy of the page in the system, even though multiple processors in the same SMP node may be accessing the page. Bypassing the software DSM layer for single-writer pages allows intra-node sharing to occur using efficient cache-coherent hardware.

Conventional software DSM systems like Munin are incapable of realizing the potential reduction in software DSM overhead for single-writer pages. Since Munin enforces consistency eagerly, every release operation will naively force the software DSM layer to perform coherence on all modified pages. For single-writer pages, such coherence operations are unnecessary because only one SMP node requires the most up-to-date version of the page. Moreover, the problem is not confined to eager release consistent systems–naive implementations of lazy release consistency [16] for DSMPs will suffer a similar problem.[4]

MGS exploits the weaker consistency guarantees needed by single-writer pages to reduce inter-node communication. An extension to the basic Munin protocol in MGS, called the *single-writer optimization*, monitors sharing patterns at runtime on a per-page basis, and dynamically identifies single-writer pages. MGS relaxes the enforcement of coherence for single-writer pages until the single-writer sharing pattern is broken. Specifically, the single-writer mechanism in MGS consists of three components:

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 be detected at a page's home SMP where the page directory can be consulted. Each time an SMP, known as the client SMP, performs a release and sends a request to the home SMP 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 SMP receives a request for coherence, it initiates invalidation. For those pages that meet the single-writer condition, the home SMP instead responds to the client with a special message indicating that the coherence policy on the page should be relaxed. The client SMP 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 SMP are ignored by the software shared memory layer. At the home SMP, the directory is marked to indicate the page is in the single-writer mode.

Reverting to Normal Coherence. The system must revert to normal coherence when a processor on another SMP tries to access the page (we will call this SMP the "3rd-party SMP"), thus violating the single-writer condition. A page fault will occur on the 3rd-party SMP since in the single-writer mode, there is only one outstanding copy of the page. During this page fault, the home SMP consults the page directory as usual. For pages in the single-writer mode, the home SMP defers service of the page fault and instead initiates invalidation. When the invalidation completes, the contents of the single-writer copy returns to the home SMP and restores the home copy to a coherent state. Finally, the 3rd-party SMP page fault is serviced.

[4]In an LRC system, consistency occurs during acquires. For single-writer pages, an LRC system will communicate with the home node on every acquire to determine whether coherence is necessary before the acquire operation is allowed to complete. While single-writer pages will not require coherence under lazy RC as they do under eager RC, the necessity to contact the home node for every acquire will still introduce significant software overhead thus degrading performance.

3.2.4 Page Cleaning

The page cleaning mechanism maintains a single view of coherent data between the hardware and software shared memory layers. Because of hardware caching, the contents of a page in the physical memory of an SMP may not represent a coherent version of the page data. For instance, there may be one or more cache lines in the page that are dirty in a processor's cache somewhere in the SMP. 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 SMP occurs through a network interface. Such interfaces typically perform data transfer by using DMA that is not coherent with respect to the processor caches. For the data transfer to see coherent data, all hardware-distributed copies need to be localized.[5]

MGS employs an all-software data localization technique, 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.

3.3 Putting it All Together: Protocol Engines

The MGS multigrain shared memory protocol, including all the mechanisms described in Section 3.2, is implemented by three software protocol engines: the Local Client, the Remote Client, and the Server. The Local Client implements the TLB consistency protocol and the client-side protocol for requesting page data from a remote SMP. The Remote Client performs page invalidation on a client SMP. And the Server handles the server-side protocol for page replication and release operations. Figures 21 and 22 in Appendix A show the state transition diagrams and state transition table, respectively, for all three protocol engines. Together, these two figures completely specify the MGS protocol. The interested reader is encouraged to read Appendix A for more details about the MGS protocol design. Additional details can be found in [17].

4 MGS Prototype Implementation

We performed a prototype implementation of the MGS system on the MIT Alewife machine [18], a hardware cache-coherent distributed shared memory architecture. Our prototype implements the MGS software DSM protocol, described in Section 3.2 and Appendix A, as a software runtime layer in between the application and Alewife's kernel. In our prototype, DSMP nodes are emulated using a technique we call *virtual clustering*. Virtual clustering partitions a single Alewife machine into smaller Alewife multiprocessors, and uses the MGS software DSM protocol for communication between partitions. Virtual clustering permits configuration of DSMP node size in software. Such system reconfigurability allows us to study all the architectures within a DSMP family on a single

[5]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 SMP reallocates the page to receive data from a remote SMP 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.

experimental platform (*i.e.* it allows us to turn the "knob" suggested in Figure 2). However, virtual clustering sacrifices some accuracy since the cost of inter-node messages are simulated. Later in this section, we will discuss the impact that virtual clustering has on our results.

In the rest of this section, we describe our prototype implementation of the MGS system. First, we give a brief overview of Alewife, and then we discuss three implementation issues for supporting MGS on Alewife: software virtual memory, virtual clustering, and active messages. Finally, we discuss the limitations of our implementation and how these limitations impact the results reported in Section 5.

4.1 The Alewife Multiprocessor

Alewife is a distributed memory multiprocessor that supports the shared memory abstraction in hardware. An Alewife machine consists of a number of homogeneous processing nodes connected in a 2-D mesh topology. Each Alewife node consists of a modified SPARC integer core, a floating point unit, 64K-bytes of static cache RAM, 8M-bytes of dynamic RAM, a 2-D mesh routing chip, and the CMMU, Communications and Memory Management Unit. Alewife supports sequential consistency, and maintains cache coherence using a single-writer write-invalidate cache coherence protocol. Also, Alewife provides a fast user-level messaging interface with DMA capability [19]. DMA data in messages are locally coherent.

4.2 Support for MGS

4.2.1 Software Virtual Memory

MGS requires a virtual memory system in order to implement software DSM. Alewife, however, is a single-address space machine and does not support virtual memory. Our MGS prototype performs address translation in software. The compiler identifies which memory accesses require translation and emits code in-line prior to these accesses to handle translation. The in-lined code reads a page table entry from the processor's local page table, and checks access rights in addition to forming a physical address. Accesses that violate access rights trap into a fault handler.

To minimize the runtime overhead of software address translation, only two types of accesses are translated in MGS: pointer dereferences and accesses to elements of distributed arrays. All other accesses, including instruction fetches, stack accesses, and local variable accesses, are unmapped and incur no translation overhead. Translation for pointer dereferences is slightly more expensive than translation for distributed arrays. Because pointers can point to both mapped and unmapped objects in memory (whereas distributed arrays are always mapped), extra overhead to translate pointer dereferences is necessary to determine whether a pointer points to a mapped or unmapped object. This distinction can be made easily at runtime because MGS places mapped and unmapped objects in disjoint parts of Alewife's address space.

Since software translation does not happen atomically, it is possible for an invalidation to occur in between the translation lookup and the data access. To prevent this from happening, the translation code includes markers that indicate a processor is in a translation critical section. A request to invalidate a mapping will interrupt the processor that owns the mapping; the interrupt handler checks to see if this processor is in a translation critical section. If so, the processor's trap return PC is rolled back to the beginning of the critical section (the translation code is reentrant).

4.2.2 Virtual Clustering

One way to prototype the MGS system is to build a DSMP by coupling several nodes together across a local area network in which each node is an Alewife machine. Unfortunately, the Alewife system does not support networking–there is no network interface hardware, nor does Alewife's kernel support any of the communication interfaces required for local area networks.[6] Instead, we implement MGS on a single Alewife machine and rely on a software technique known as *virtual clustering* to emulate a DSMP.

Virtual clustering logically partitions an Alewife machine into virtual SMPs by disallowing the use of hardware-supported shared memory between logical partitions. The partitioning can be easily enforced: since a processor cannot access what it cannot name, shared memory traffic can be contained within virtual SMP nodes simply by disallowing processors from mapping pages across virtual SMP nodes. Such a page mapping policy will force any processor that wants to access pages on remote nodes to trap into MGS software and faithfully run the identical software DSM code that would run on an actual DSMP.

While our virtual clustering approach accurately accounts for software shared memory overheads, it does not address the overheads associated with communication across a local area network since communication between virtual SMP nodes in our MGS prototype uses the fast Alewife messaging mechanisms. To better model the cost of inter-SMP communication on an actual DSMP, we artificially delay all inter-SMP messages. Whenever a processor sends a message to a remote virtual SMP node, the message is placed on a software queue and a hardware timer is set for some amount of delay. When the timer counts to zero, a timer interrupt handler dequeues the message and sends it through the Alewife network. By artificially delaying messages, we model the cost of inter-SMP communication as a fixed latency. Our implementation of MGS does not account for contention neither in the LAN, nor in each SMP's interface to the LAN.

4.2.3 Active Messages

MGS relies heavily on the active message layer supported by Alewife for efficient communication. Two architectural features make active messages particularly efficient. First, Alewife provides support for DMA bulk data transfer in messages. All page-size data is transferred using DMA thus relieving the processor of per-byte transfer overheads. Second, there are four hardware contexts in the Alewife integer core that accelerate active message handler invocation. The hardware contexts eliminate the need to save and restore registers on handler entry and exit. In addition, preallocation of thread meta-data structures such as stacks and task blocks to each of the hardware contexts allows incoming messages to execute as handlers immediately. Handler invocation becomes more expensive only when there are no free hardware contexts on message entry.

4.3 Limitations of the Implementation

Our prototype implementation of the MGS system accurately reflects the behavior of a DSMP in that the software DSM protocol of the MGS system is fully implemented in a software runtime layer. There is no emulation or simulation whatsoever in this software runtime layer–all code is faithfully

[6] Alewife relies on a host workstation for communication with clients in the local area.

executed on the Alewife hardware as it would in an actual DSMP, including all required user-kernel address space crossings. However, two elements of our implementation potentially impact the accuracy of the results reported in Section 5: using virtual clustering to emulate DSMP nodes, and using the Alewife machine as the node architecture.

Virtual clustering enables reconfiguration of the DSMP node size in software, thus facilitating the comparison between different DSMP architectures. However, virtual clustering sacrifices some accuracy because the inter-node communication cost is modeled as a fixed latency. As a result, contention in the inter-node network and at the inter-node network interfaces is not modeled. A DSMP with an actual inter-node network would exhibit lower performance than our results indicate if contention increases the actual inter-node communication latency beyond the fixed latency used in our experiments. In Section 5.6, we quantify the impact of a fixed inter-node communication latency assumption on our results by varying the cost of messaging between DSMP nodes and observing the sensitivity of our results to changes in inter-node communication latency.

Using Alewife as the node architecture has two consequences. First, our prototype inherits the support for fast interrupts provided by Alewife. On Alewife, a message arrival interrupt can be serviced in 5-10 μsec, roughly an order of magnitude faster than on a commercial operating system. Therefore, our results represent a "best-case" level of performance since an implementation of our MGS design on a commercial operating system would achieve noticeably lower performance due to the higher cost of interrupts. We note, however, that the impact of costly interrupts on a commercial operating system can be minimized by modifying our MGS design to reduce the frequency of interrupts. Prior work [20, 8] has investigated polling techniques to eliminate interrupts for message invocation and TLB invalidation events. We believe that results similar to those reported in Section 5 can be achieved on a commercial operating system if existing techniques for reducing the frequency of interrupts are integrated into our MGS design.

Finally, Alewife is a DSM, not a bus-based multiprocessor. Since DSMs offer superior scalability, Alewife nodes permit larger node sizes than bus-based nodes. Our choice of a scalable node architecture reflects the trend in current server-class SMPs towards higher scalability. For example, SUN Microsystem's Enterprise SMP servers [5] employ switched interconnect instead of a bus (though the cache-coherence protocol in this machine is still broadcast-based). Also, the SGI Origin [10] is a DSM marketed as a "Scalable SMP" that will replace SGI's bus-based servers. Because we use a scalable node architecture, our results represent the performance of DSMPs built using future server-class SMPs. DSMPs built with bus-based SMPs may not achieve the same level of performance indicated by our results, particularly for DSMPs with large node sizes, since bus-based architectures suffer contention across the bus interconnect.

5 Results

We first present measurements that show the cost of primitive MGS operations in Section 5.1. Section 5.2 describes eight shared memory applications used in our application study, and Section 5.3 presents extensive results showing the performance of these applications on our MGS prototype. Section 5.4 examines the bottlenecks preventing higher performance on MGS for the four most difficult applications, and discusses how application restructuring can relieve these bottlenecks. Section 5.5 examines the performance of the restructured applications. Finally, Section 5.6 examines the sensitivity of our experimental results to inter-node network latency.

Hardware Shared Memory		Software Shared Memory	
Cache Miss Local	11/12	TLB Fill	2302/3590
Cache Miss Remote	38/38	Page Fault	11772/21956
Cache Miss 2-party	42/43	Upgrade Fault	12441
Cache Miss 3-party	63/66	Page Fault, Single-Writer	29353/35293
Remote Software	425/707	Single-Writer Transition	9992
Software Virtual Memory		Release (2 writers)	33424
Distributed Array Translation	16	Release (3 writers)	33516
Pointer Translation	23		

Table 1: Shared Memory Costs on MGS. Whenever two numbers are given in the second column, the first is for read operations, and the second is for write operations. All numbers are in cycles assuming a 20 MHz clock frequency.

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

5.1 Micro Measurements

Table 1 shows the cost of performing some basic shared memory operations on MGS. These measurements were taken on an Alewife machine running at 20 MHz. There are three groups of measurements. The first group measures the cost of hardware shared memory on Alewife. These latencies represent the penalty for various types of cache misses. They do not include the overhead of software address translation. The entry labeled "Remote Software" reports the cost of a miss to a cache line under software directory control (the Alewife cache-coherence protocol, known as LimitLESS [21], provides a fixed number of directory pointers in hardware and traps into software for pointer overflow).

The second group of measurements shows the cost of software address translation. Translation for both distributed array objects and general pointers are shown. Finally, the last group of measurements report the cost of MGS' software coherence protocol. All measurements were taken assuming a 1K-byte page size and a 1000 cycle delay (50 μsec) for communication between SMPs.

5.2 Applications

Table 2 lists the applications used to study the performance of our MGS prototype. There are eight applications in total: Jacobi, Matmul, FFT, Gauss, Water, Barnes-Hut, TSP, and Unstructured. The first four are small scientific kernels. Jacobi performs an iterative relaxation over a two-

Application	Seq	Sp1	SVM-Seq	Ovhd	SVM-Par	Sp2
Jacobi[7]	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
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 3: 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.

dimensional grid, Matmul multiplies two dense matrices, FFT computes a one-dimensional fast Fourier transform, and Gauss performs Gaussian elimination on a matrix. The next two are codes from the SPLASH-I benchmark suite [22]. Water is a molecular dynamics code, and Barnes-Hut is a hierarchical N-body simulation. Both simulate the motion of particles in three-dimensional space. TSP is the traveling salesman problem. It uses a branch and bound algorithm to prune its search and a centralized work queue to distribute work. Finally, Unstructured is a computation over an unstructured mesh [23]. The computation resembles solving Eular equations on unstructured meshes, but does not actually produce meaningful numeric results.

The last two columns of Table 2 specify the problem size used in the experiments for each application, and the number of lines of C code, respectively. The two applications, Water-Kernel and Unstructured-Kernel, are variants on the main Water and Unstructured applications and will be described in Section 5.4.

Table 3 provides baseline performance numbers for our applications on Alewife without the overheads of software shared memory that would be incurred by a DSMP. 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.

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 4.2.1. "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. The column labeled "SVM-Par" reports the running time on a 32-node Alewife machine. These parallel performance numbers include the overhead of software

[7]Jacobi's problem size was not able to fit in the memory of a single Alewife node; therefore, we ran the problem on 4 nodes for both the "Seq" and "SVM-Seq" columns, and extrapolated the single node numbers by assuming linear speedup from 1 to 4 processors.

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. Therefore, these numbers represent the performance on a hardware DSM (modulo software address translation), and is what we compare DSMP performance against in the results presented in Section 5.3.

Finally, the last column in Table 3, 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 software virtual memory overhead, which we expect to parallelize perfectly, does not increase the computation-to-communication ratio of our applications such that they become embarrassingly parallel.

5.3 Application Results

The results for the individual applications appear in Figures 6 through 13. All measurements were performed on our MGS prototype, running on a 32-node 20 MHz Alewife machine with a page size of 1K-bytes. The inter-SMP communication latency used is 1000 cycles (50 μsec) in all cases (we will study the impact of varying the inter-SMP communication latency in Section 5.6). We present the data using the performance framework discussed in Section 2.3. For each application, we observe the application's execution time (along the y-axis) on a 32-processor DSMP as SMP node size is varied from 1 to 32 in powers of 2 (along the x-axis).

Each execution time data point in Figures 6 through 13 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 cycles spent usefully in user code, but it also counts cycles spent in software address translation and Alewife cache-coherent shared memory stall. The synchronization components include both the overhead of executing synchronization code and waiting on synchronization conditions. Finally, the 32-processor SMP node size data points (the rightmost bars in Figures 6 through 13) are exactly the runtimes reported in the "SVM-Par" column of Table 3. As described earlier, these bars represent performance on an all-hardware DSM, so there is no MGS component. In addition, the synchronization components have been folded into the user component because the native Alewife experiments use a different synchronization library than the DSMP experiments for which cycle counting was not instrumented.

Based on the performance levels achieved, we group the applications into three categories: coarse-grain, medium-grain, and fine-grain applications.

5.3.1 Coarse-Grain Applications

Jacobi, Matmul, FFT, and Gauss, presented in Figures 6, 7, 8, and 9, respectively, are coarse-grain applications. Using the performance metrics introduced in Section 2.3, these applications all have a small multigrain potential and a small breakup penalty. The small multigrain potential indicates that very little benefit is experienced as SMP node size is increased and more hardware cache-coherent shared memory is provided in each SMP node. The small breakup penalty indicates that DSMPs closely match the performance of all-hardware DSMs 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*.

All four applications exhibit coarse-grain sharing: each processor performs large amounts of independent work before communicating with other processors. Coarse-grain 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. This observation is consistent with the flat performance profiles in Figures 6 through 9. A flat performance profile signifies that the application is insensitive to the underlying implementation of shared memory.

There are two anomalies in our results which warrant explanation. First, Jacobi exhibits a negative breakup penalty (*i.e.* the DSMP outperforms the all-hardware DSM). 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. Hardware DSMs must move this data one cache-line at a time, which is less efficient. Second, the multigrain potential in Gauss is negative. This is an artifact of Alewife's LimitLESS cache-coherence protocol which supports directory pointer overflow in software. 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 SMP node size is increased from 4 processors to 8 processors. Because the cost of LimitLESS is quite high, as documented in Table 1, DSMPs with smaller SMP nodes (where directory pointer overflow is impossible) outperform those with larger SMP nodes.

5.3.2 Medium-Grain Applications

Water and Barnes-Hut, presented in Figures 10 and 11, respectively, are medium-grain applications. In contrast to the coarse-grain applications, medium-grain applications exhibit a performance profile that has 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 DSMPs because it means that supplying hardware-supported cache-coherent shared memory between more processors (*i.e.* building larger SMP nodes) improves performance. This suggests that DSMPs 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 DSMPs and all-hardware shared memory systems; therefore, on these applications, all-hardware DSMs hold a performance advantage over DSMPs.

As Figure 10 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. In Water, the computation of inter-molecule force interactions involves frequent write sharing between all processors. On DSMPs, this results in significant inter-SMP communication, and thus high MGS overhead. Furthermore, the write sharing occurs at a granularity that is smaller than a page; therefore, there is significant false sharing that adds to the MGS overhead. As SMP node size increases, the MGS overhead is partially alleviated since a larger fraction of the write sharing is handled by cache-coherent shared memory. However, significant MGS overhead still occurs even at the largest SMP node sizes due to the all-to-all nature of write sharing in Water.

The results for Barnes-Hut appear in Figure 11. As the figure shows, the most significant source of slowdown in Barnes-Hut is lock overhead. The poor lock performance in Barnes-Hut is due to an effect that we call *critical section dilation*. In Barnes-Hut, a common operation on each processor is to obtain a lock, write a value in some data structure, and then relinquish the lock. On a hardware

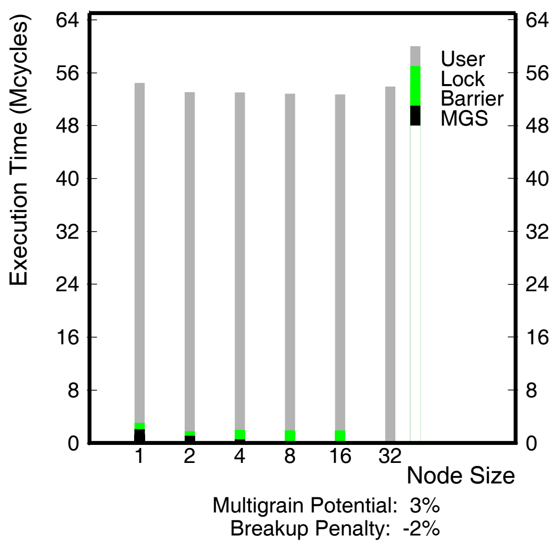

Figure 6: Runtime breakdown for Jacobi.

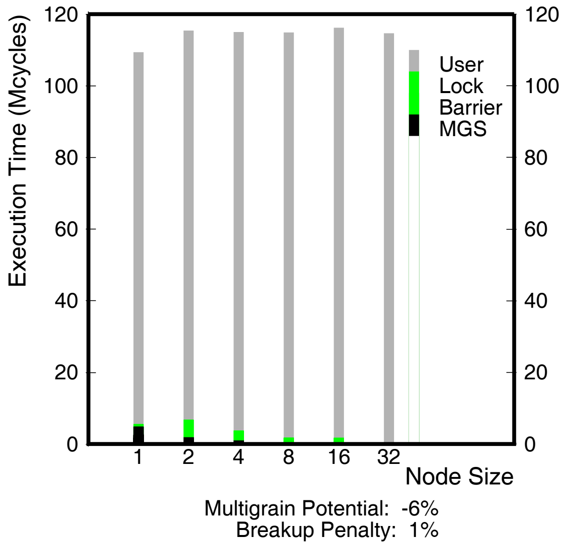

Figure 7: Runtime breakdown for Matrix Multiply.

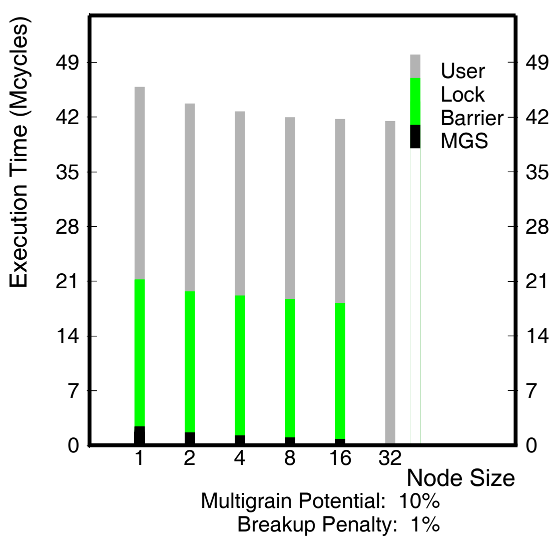

Figure 8: Runtime breakdown for FFT.

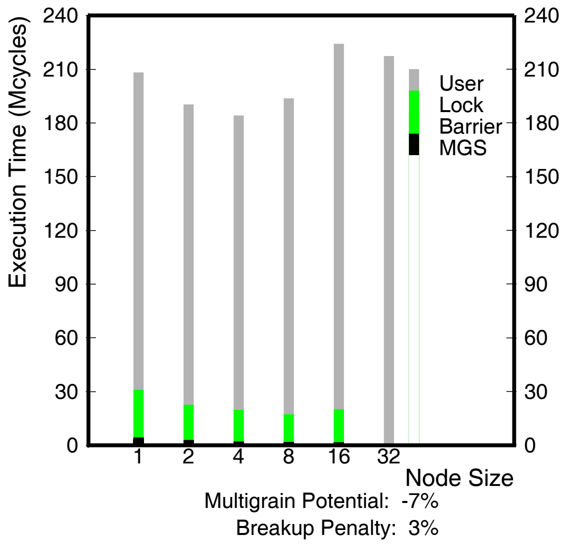

Figure 9: Runtime breakdown for Gauss.

DSM, these operations complete with very low overhead. However, on a DSMP, a TLB fault or a page fault (or both) can be suffered on the updated location. When they occur, such software overhead significantly increases the cost of the locking operation, and thus dilates the length of the critical section leading to increased lock contention.

In addition to lock overhead, Barnes-Hut also exhibits significant barrier 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 [24] for details), the technique is not perfect thus accounting for some of the barrier overhead. Load imbalance can also arise due to MGS overhead. Processors that provide software DSM service for "hot" pages will carry a disproportionate fraction of the MGS protocol processing load thus contributing to load imbalance.

5.3.3 Fine-Grain Applications

TSP and Unstructured, presented in Figures 12 and 13, respectively, are fine-grain applications. These applications have a similar performance profile as compared with medium-grain applications (large multigrain potential and a large breakup penalty). A key difference, however, is the breakup penalty is so large that the DSMPs do not achieve any effective speedup, even as SMP node size is increased. The DSMP speedups for TSP and Unstructured are all below 2, with TSP exhibiting slowdown in the worst case.

Figure 12 shows that TSP suffers from extremely high lock overhead. The source of lock overhead in TSP is the 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. Because the work pool uses locks to enforce mutual exclusion, the overhead associated with the work pool shows up as lock overhead due to critical section dilation which is severe because there is only a single work pool for the entire machine. Notice, however, that despite the contention on the centralized work pool data structure, the all-hardware DSM system manages to achieve decent performance nonetheless. This speaks volumes about the robustness of hardware DSMs on applications with poor locality characteristics.

The other fine-grain application is Unstructured, whose results appear in Figure 13. Unstructured is by far the most difficult application to achieve high performance on DSMPs because of its highly irregular data access patterns. The application performs a computation on an undirected graph which is read from an input file. Because of the graph's irregular nature, runtime preprocessing techniques [25] 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 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 whose edges are assigned to different processors by the runtime preprocessing phase).

The poor performance of Unstructured on DSMPs is attributable to all three overheads reported in Figure 13: 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. Barrier overhead is due to an imbalance in the schedule of edge computations inside the edge loops. While the runtime

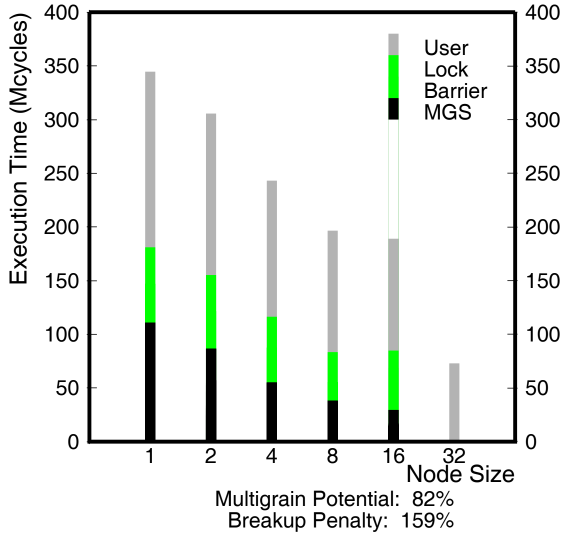

Figure 10: Runtime breakdown for Water.

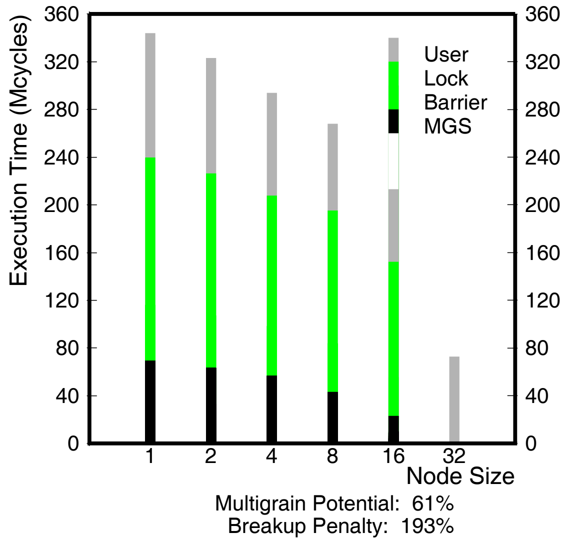

Figure 11: Runtime breakdown for Barnes-Hut.

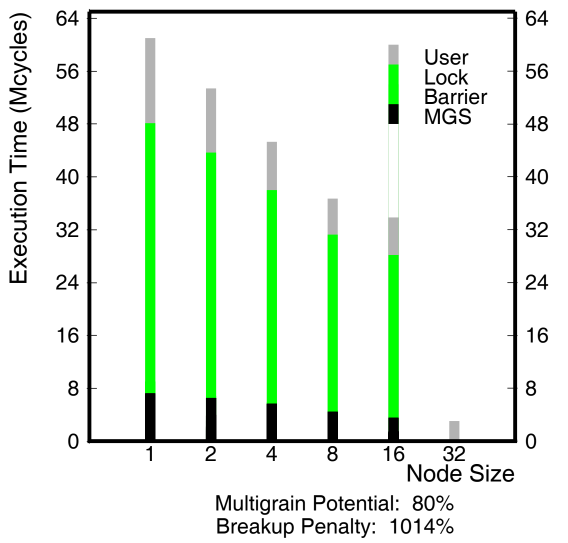

Figure 12: Runtime breakdown for TSP.

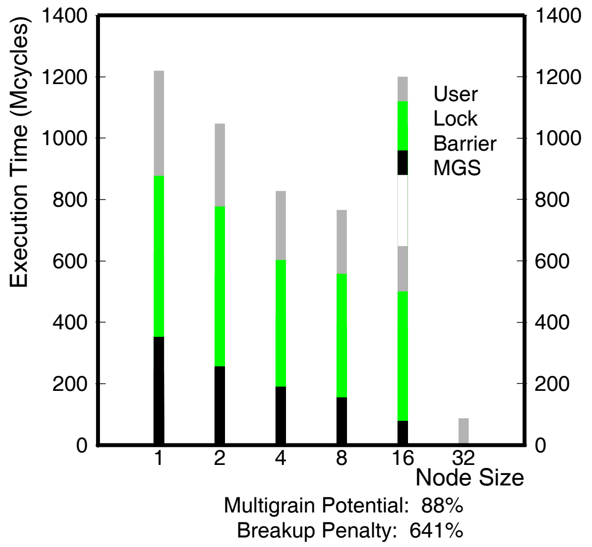

Figure 13: Runtime breakdown for Unstructured.

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

Table 4: Summary of performance bottlenecks and transformations.

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.

5.4 Application Transformations

In this section, we further examine the medium-grain and fine-grain applications studied in Section 5.3. In particular, we propose transformations to address the bottlenecks that limit performance, and apply them to the applications by hand. In most cases, the transformations are similar to optimizations performed by existing compilers; therefore, we expect several of our transformations can be automated in an optimizing compiler. In the interest of space, we only provide a brief description of the application transformations. The interested reader is referred to [17] for a detailed description of the transformations and a discussion on how existing optimizing compilers implement such transformations.

Table 4 lists the four medium-grain and fine-grain applications, the bottlenecks that limit performance, and the transformation(s) that relieves each bottleneck. In the case of Water and Unstructured, we only study a kernel from the original application where significant performance bottlenecks exist. Water-Kernel executes the force interaction computation where Water spends most of its execution time and where the all-to-all write sharing patterns occur (see Section 5.3). Unstructured-Kernel executes a single edge loop. The baseline performance of these kernels on Alewife can be found in Table 3.

We address the data locality problems in the Water workload by performing a loop tiling transformation. Loop tiling groups molecules together into tiles, and restructures the loops that perform the force interactions between molecules such that the order in which the interactions are performed are blocked according to the tiles. This increases data locality since all the interactions associated with a particular tile are computed before considering new tiles.

Three transformations are proposed for the Barnes-Hut workload in Table 4, all to address the critical section dilation problems discussed in Section 5.3. One source of critical section dilation occurs on a freelist data structure that is protected by a single lock. We remove the lock and allow processors to allocate concurrently off the freelist by statically assigning freelist entries to processors in an interleaved fashion. Next, we physically distribute the freelist so that each processor is allocated entries that are contiguous in memory. This removes false sharing communication later when the entries are manipulated within critical section code (and thus relieves the dilation to those critical sections). Finally, there are a few instances in Barnes-Hut where a large number of

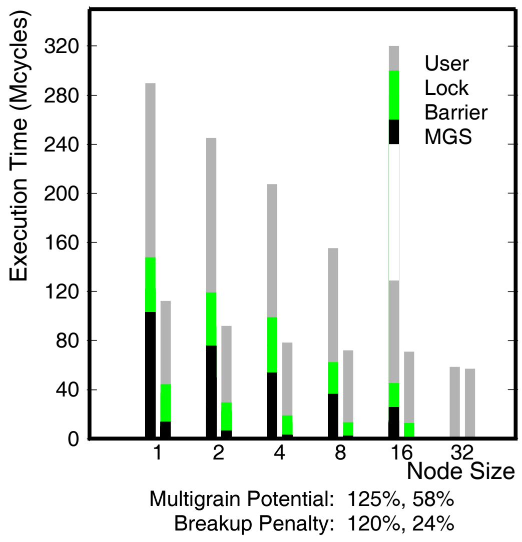

Figure 14: Transformation results for Water-Kernel.

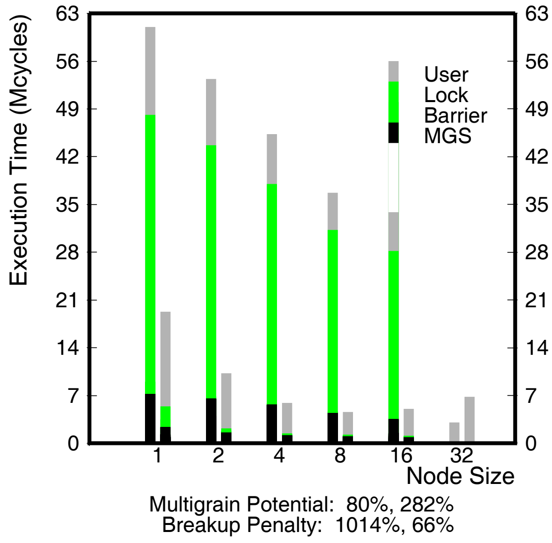

Figure 15: Transformation results for TSP.

pages are updated right before a critical section is entered. As a result, the software coherence for all these extra pages are resolved within the critical section. We add a release right before entering the critical section so that the overhead for processing the extra pages is incurred outside of the critical section.

The contention problems on the work pool data structure in the TSP workload are addressed by replacing the centralized work pool with a distributed work pool, one locally for each SMP. Software shared memory is avoided entirely for local work pools as accesses to the local work pool are supported by hardware cache-coherent shared memory. 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 off-loaded to the local work pools.

To address the data locality problems in Unstructured (which also contribute to its critical section dilation problems), we perform a tiling transformation. This transformation is similar to the one performed for Water, except we must compute the tiles and the tile interaction schedule at runtime since the interactions are computed according to an unstructured mesh whose topology is known only at runtime (in Water, the tile interaction schedule can be computed statically). Also, we address the load imbalance problems in Unstructured by spreading the interaction assignments evenly across processors in the same SMP. While this addresses load imbalance between processors within an SMP, load imbalance between SMPs is not addressed. We chose to sacrifice load balance for data locality between SMPs.

5.5 Transformation Results

Figures 14 through 17 present detailed results for the application transformations using our performance framework. The presentation format is identical to the one used in Section 5.3, except that we plot the results of both the original application and the restructured application, side by side.

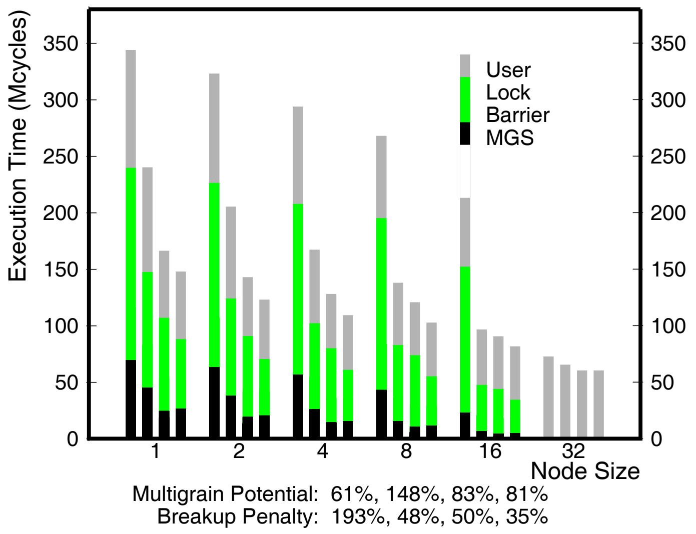

Figure 16: Transformation results for Barnes-Hut.

Figure 14 shows the detailed results for the Water-Kernel workload, before (left set of bars) and after (right set of bars) the loop tiling transformation is applied. As demonstrated by the significant reduction in the MGS overhead component, the loop tiling transformation is extremely effective in improving data locality in Water's force interaction computation. Furthermore, the transformation becomes even more effective on larger SMP nodes since more computation within SMP nodes is possible before inter-SMP communication must occur to communicate tiles.

Figure 16 shows the results for transformations on Barnes-Hut. The four bars at each SMP node size report the performance observed on the original application as the three transformations are applied incrementally. The first transformation to allow concurrent freelist allocation (2nd set of bars) significantly improves performance. The second transformation to eliminate false sharing (3rd set of bars) also significantly improves performance, though its effects are most pronounced at smaller SMP node sizes. Finally, the transformation to reduce critical section dilation by adding releases (rightmost set of bars) produces the least gain in performance.

TSP, shown in Figure 15, displays the greatest gains in performance due to its transformation. Figure 15 shows that the enormous overheads associated with the centralized work pool can be mostly eliminated by using a distributed data structure instead. TSP exhibits some unexpected behavior at SMP node sizes 16 and 32—performance decreases with increasing SMP 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 SMP node, the global work pool serves no purpose (since it is meant to distribute work between SMP nodes). However, it 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 17 shows transformation results for the Unstructured-Kernel workload. The first transformation (2nd set of bars) successfully improves data locality as demonstrated by the reduction in the MGS overhead component. However, the improvement in data locality comes at the expense of load balance. Load imbalance increases at smaller SMP node sizes because the

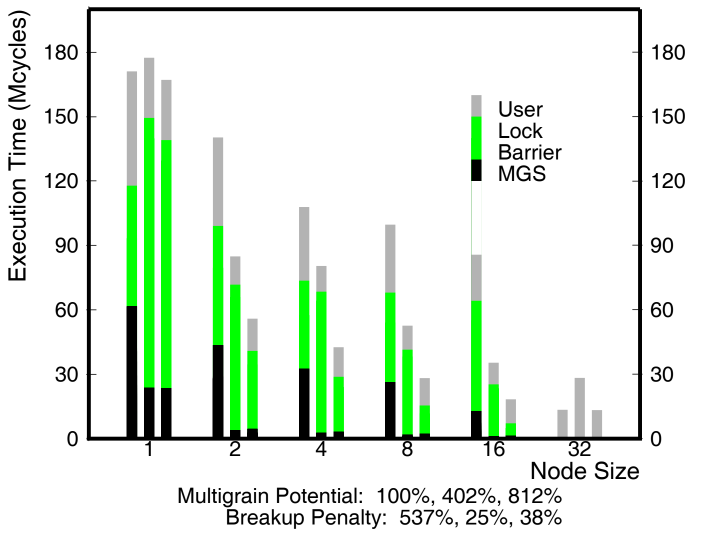

Figure 17: Transformation results for Unstructured.

smaller tiles used for smaller SMP nodes lead to a greater variability in the number of interactions between two tiles. The transformation actually degrades performance at SMP node sizes of 1 and 32. 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 SMP node sizes except for the SMP node size 1. Our runtime load balancing transformation only attempts to load balance between processors in the same SMP, since we didn't want to sacrifice data locality between SMPs. When there is only a single processor per SMP node, the load balancing transformation is useless.

5.6 Network Latency Sensitivity

Our implementation of MGS emulates an inter-SMP network using a fixed messaging latency. For the results reported in Sections 5.3 and 5.5, we set this inter-SMP latency to 1000 Alewife cycles, or 50 μsec. Our approach has two short-comings. First, by assuming a fixed communication latency, our results do not reflect the impact of contention in the inter-node network and at the inter-node network interfaces. Second, by using a latency of 50 μsec, our results only reflect the performance of DSMPs with aggressive networks. While a 50 μsec one-way messaging latency is achievable (*e.g.* [3] reports a latency of 33 μsec over ATM), such low latency is possible only on high-performance networks. In this section, we explore the impact that higher communication latency, due to either contention or a slower network, has on our results. Our study exploits the hardware timers used to set inter-SMP communication latency (see Section 4.2.2) to change the cost of inter-SMP messages.

Figures 18–20 show the impact of varying inter-SMP communication latency for the three applications, Jacobi, Water, and Water-Kernel with tiling, respectively. Execution time (in millions of cycles) is plotted on the Y-axis against inter-SMP communication latency (in thousands of cycles) on the X-axis for three different node sizes–1, 4, and 16 processors. In all three graphs, the inter-SMP communication latency is varied between 95 cycles (the minimum latency for Alewife

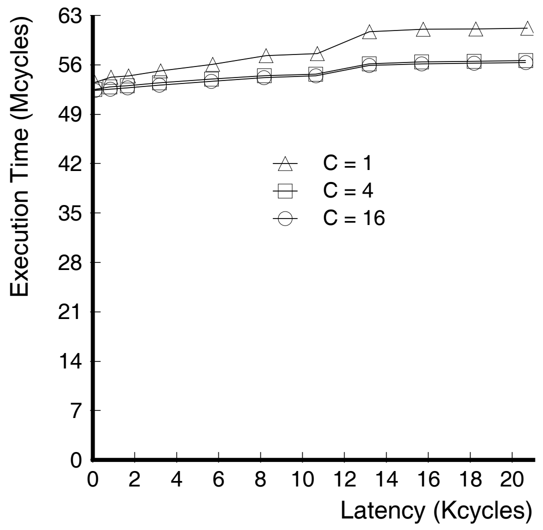

Figure 18: Latency sensitivity results for Jacobi.

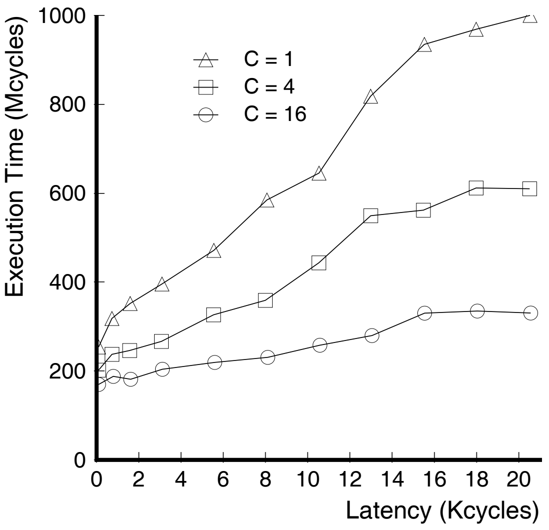

Figure 19: Latency sensitivity results for Water.

messages) and 20,000 cycles, or between 4.75 μsec and 1 *m*sec. In the following discussion, we examine the sensitivity of application performance to changes in the inter-SMP communication latency beyond the 50 μsec baseline latency used in our experiments.

Figure 18 shows that Jacobi is insensitive to variations in inter-SMP communication latency since the slope of the three curves are small. When the inter-SMP communication latency is 10,000 cycles or 500 μsec (a factor 10x higher than our baseline latency), execution time increases by only 5.8%, 3.1%, and 3.2% for 1-, 4-, and 16-processor node sizes, respectively, over the execution time which assumes the baseline network latency. Because Jacobi performs very little communication, even very large changes in inter-SMP communication latency only have a small impact on overall performance. Therefore, both contention effects and higher-latency networks are unlikely to significantly alter the results reported for coarse-grain applications like Jacobi in Section 5.3.

Compared to Jacobi, the Water workload displays much higher sensitivity to inter-SMP communication latency due to its higher communication volume, as indicated by the steep slopes of the three curves in Figure 19. When the inter-SMP communication latency is 2,000 cycles or 100 μsec (a factor 2x higher than our baseline latency), execution time increases by 12.47%, 8.31%, and 12.41% for 1-, 4-, and 16-processor node sizes, respectively, over the execution time which assumes the baseline network latency. At an inter-SMP communication latency of 10,000 cycles or 500 μsec, execution time increases by 83.4%, 80.13%, and 42.23%, respectively. At an inter-SMP communication latency of 20,000 cycles or 1 *m*sec, execution time increases by 184.2%, 147.9%, and 82.1%, respectively. These results suggest that higher latency due to contention effects or slower networks will alter the results reported for medium-grain applications like Water in Section 5.3. To maintain reasonable agreement with our results (say within 10%), fast networks that have a one-way message latency no greater than 2x of our baseline latency are necessary, with ample bandwidth to avoid contention. If slower networks are used, however, we observe that the disagreement with our results will be much less for DSMPs with large SMP nodes. Figure 19 demonstrates that DSMPs with

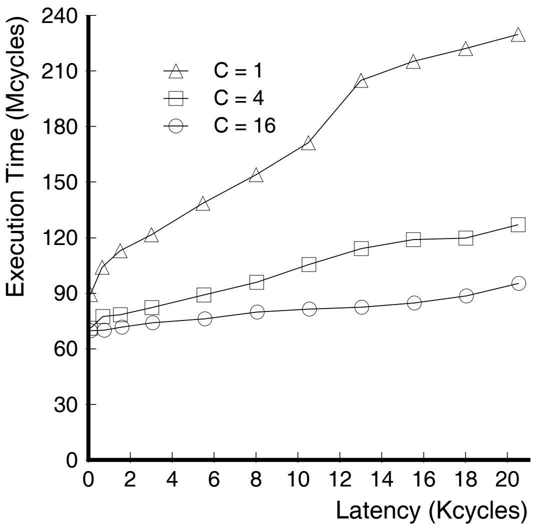

Figure 20: Latency sensitivity results for Water-Kernel with tiling.

large SMP nodes are much less sensitive to latency than DSMPs with small SMP nodes since the inter-SMP communication volume decreases significantly with increasing node size.

Finally, Figure 20 shows the latency sensitivity of the tiled Water-Kernel code. We can see from the figure that the tiled code is less sensitive to inter-SMP communication latency than the original Water application. At an inter-SMP communication latency of 10,000 cycles or 500 μsec, execution time increases by 51.6%, 34.6%, and 13.6% for 1-, 4-, and 16-processor node sizes, respectively, over the execution time which assumes the baseline network latency. At an inter-SMP communication latency of 20,000 cycles or 1 $m$sec, execution time increases by 103.4%, 62.0%, and 33.0%, respectively. Since the locality transformations reduce inter-SMP communication volume, they also reduce the application's sensitivity to inter-SMP communication latency.

6 Related Work

The work presented in [4] describes our early version of the MGS system which forms the basis for this paper. Since then, several multigrain shared memory systems that exploit SMP nodes have been built. Compared to MGS, these systems more deeply explore the implementation issues associated with industry-grade operating systems since they all use commercial SMPs. On the other hand, the MGS work deals more deeply with the behavior of applications in a multigrain environment, since it is the only work to compare different DSMP configurations, all-software DSMs, and all-hardware DSMs on a single experimental platform.

Most recently, the work in [6] describes an implementation of the home-based lazy release consistency protocol (HLRC) [26] on a cluster of four 4-way Intel Pentium Pro SMPs. Compared to MGS, this system uses a more aggressive software DSM protocol that enforces coherence lazily at acquire operations, and therefore generates fewer inter-node messages. The main contribution of the work lies in an extension of LRC for SMPs that employs a two-level timestamp, one for each processor and one for each node, to maintain the causal relationship between acquire and release operations required by LRC. By recording both per-processor and per-node timestamps, the system is able to more accurately determine the set of pages that require invalidation at an acquire operation, thus avoiding unnecessary invalidations that would occur if timestamps were maintained on a per-node basis only. In the MGS system, our separate DUQ lists serve a similar purpose to "refine the invalidation set" for eager release consistency protocols (see Section 3.2).

The Cashmere-2L system [7] is a cluster of eight 4-way DEC AlphaServer SMPs. Again, as in the HLRC SMP system, Cashmere-2L implements a more aggressive software DSM protocol than MGS that is modeled after LRC. One crucial difference is that Cashmere-2L employs the Memory Channel [27] as the inter-node network. The Memory Channel provides hardware support for low-latency remote writes between nodes, thus client nodes in the Cashmere-2L protocol can perform updates to home node copies with very low overhead and without interrupting the home node. In contrast, the MGS system targets less aggressive networks. The Cashmere-2L protocol also employs an "exclusive mode" for pages cached by a single SMP node. This protocol optimization provides the same communication reduction benefits for LRC protocols as our single-writer optimization provides for eager RC protocols (see Section 3.2).

In [8], an implementation of the Shasta system [28] on a cluster of four 4-way DEC AlphaServer SMPs is described. The base Shasta system uses software address translation (similar to our implementation of software virtual memory described in Section 4.2.1 but with much less overhead) to enable a variable software DSM coherence unit size. One major challenge in extending the base Shasta system to SMPs lies in addressing the race condition that occurs between translation code and page invalidation operations. The Shasta SMP solution explicitly synchronizes invalidation and translation code to eliminate the race condition. Our implementation of MGS encounters the same problem, but our solution detects the race condition and recovers via roll back (see Section 4.2.1). Compared to MGS, the Shasta SMP system implements a more eager release consistency protocol that propagates updates before release operations, much like in a hardware DSM. However, Shasta SMP is less sensitive to false sharing than MGS despite such eager consistency because of the variable-sized coherence units supported in Shasta. As with Cashmere-2L, another difference is that Shasta SMP employs the Memory Channel interconnect for low-latency communication. The Shasta SMP system also minimizes the impact of expensive interrupts by polling for inter-node messages and mapping invalidation events. As discussed in Section 4.3, such polling techniques can benefit an MGS system targeted for commercial SMPs.

Concurrent with the original MGS work, two other multigrain systems were developed. SoftFLASH [9] built multigrain shared memory on a cluster of four 8-way SGI Challenge SMPs. Three different software shared memory protocols were implemented on this cluster. Two of the protocols, a sequential consistency protocol and an eager release consistency protocol, are the same protocols supported on the FLASH multiprocessor [29]. The third protocol is a variant of LRC. The MGS protocol is most similar to the eager release consistency protocol. Experiences on SoftFLASH demonstrate two bottlenecks that limit performance on the SGI Challenge: the high cost of TLB coherence due to expensive interrupts, and limited inter-node bandwidth. In addition to SoftFLASH, another multigrain system was built using four 4-way SMPs over an ATM network [30]. The primary focus of this work was to study prefetching techniques to hide the large latencies associated with paging between SMP nodes.

The construction of the MGS system relies heavily on software shared memory protocols for uniprocessor nodes. Many software shared memory protocols have been proposed [31, 32, 11, 2, 16, 33]). As described throughout this paper, the software DSM layer in MGS most closely resembles the Munin system [11]. Better software DSM performance has been demonstrated since Munin using lazy release consistency, most notably in the Treadmarks system [2]. As described above, several multigrain systems since MGS have employed LRC to reduce inter-node communication.

Finally, the idea to couple hardware cache-coherent shared memory with software page-based shared memory was first suggested in [34]; however, this work was purely simulation based using an extremely simple machine model. None of the design nor performance issues associated with integrating hardware and software shared memory were explored.

7 Conclusion

This paper investigates building large-scale shared memory machines using small- to medium-scale multiprocessors as the basic building block. Such multigrain systems are attractive for two reasons. First, small- to medium-scale multiprocessors are economically viable and will have an ever increasing presence in the local area environment. It simply makes sense to leverage these commodity components to build larger systems. Second, SMPs already have hardware support for shared memory. Large-scale shared memory machines built from SMPs can achieve high performance if they effectively leverage the efficient hardware mechanisms provided by each SMP.

Our work makes several novel contributions in the context of DSMP architectures. We present a fully functional design of a multigrain shared memory system, called MGS, and provide a prototype implementation of MGS on the MIT Alewife multiprocessor. We define two performance metrics, the breakup penalty and the multigrain potential, that characterize application performance on DSMPs. Finally, we conduct an in-depth application study using our MGS prototype implementation, including a study of data locality transformations to more effectively leverage the clustered nature of DSMPs. While performance evaluations have been conducted on other DSMP systems, our results are the first to explore the entire spectrum of DSMP architectures, and to provide a consistent comparison of these architectures against traditional all-software and all-hardware DSMs on a single experimental platform.

Based on the results of our experimental study, we draw several conclusions regarding the performance of DSMPs. First, applications that exhibit coarse-grain sharing patterns achieve high performance regardless of the underlying mechanisms for shared memory. All-software DSMs, DSMPs, and all-hardware DSMs offer equivalent performance on these unchallenging applications. Second, for applications with finer-grained sharing patterns, the fine-grain mechanisms provided by DSMPs within SMP nodes offer significant performance advantages. The four medium- to fine-grain applications perform between 61% and 88% faster (*i.e.* the multigrain potential ranges from 61% to 88%) on DSMPs as compared against all-software DSMs. This evidence strongly suggests that SMPs are much better building blocks than uniprocessor workstations for large-scale multiprocessors. Unfortunately, our study also shows that these applications exhibit fine-grain sharing across the entire machine. Such global sharing places a severe load on the software communication mechanisms between SMP nodes. As a result, our third conclusion is that DSMPs cannot offer nearly the same level of performance on difficult applications compared with all-hardware DSMs. In our study, hardware DSMs perform 159% to 1014% faster (*i.e.* the breakup penalty ranges from 159% to 1014%) on medium- and fine-grain applications.

In addition, our study also explores application transformations that further improve performance by exposing the clustered nature of the machine to the programmer or compiler. We find that fine-grain sharing can be confined within SMP 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 within SMP nodes. When the transformations are applied, three out of the four difficult applications exhibit breakup penalties below 40%. The other has a moderate breakup penalty of 66%. Based on these results, our fourth conclusion is that DSMPs can become competitive with all-hardware DSMs in absolute performance, even on difficult fine-grain applications. It is important to note that even after data locality transformations are applied, DSMPs still hold a significant performance advantage over all-software systems. Our results show that the multigrain potential after transformations ranges between 58% and 812%. This result leads to our last conclusion that there is something fundamental about the nature of fine-grain sharing in these applications. Our transformations do not eliminate fine-grain sharing; they only limit the extent to which fine-grain sharing occurs across the machine. Therefore, supporting such applications efficiently still requires fine-grain shared memory mechanisms.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] The Ultra Enterprise 1 and 2 Server Architecture. SUN Microsystems. Mountain View, CA, 1996.

[6] Rudrajit Samanta, Angelos Bilas, Liviu Iftode, and Jaswinder Pal Singh. Home-Based SVM Protocols for SMP Clusters: Design and Performance. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, Las Vegas, NV, February 1998. IEEE.

[7] 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 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997. ACM.

[8] Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, Las Vegas, NV, February 1997.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Donald Yeung. Multigrain Shared Memory, PhD Thesis. MIT-LCS TR-743, Massachussetts Institute of Technology, February 1998.

[18] 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.

[19] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Supercomputing Conference*, Tokyo, Japan, July 1993.

[20] Babak Falsafi and David A. Wood. Scheduling Communication on an SMP Node Parallel Machine. In *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, February 1997.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[27] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2), February 1996.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

A MGS Protocol Engine Specification

The MGS multigrain shared memory protocol, including all the mechanisms described in Section 3.2, is implemented by three software protocol engines called the Local Client, the Remote Client, and the Server. Figures 21 and 22 show the state transition diagrams and state transition table for these protocol engines, respectively, and together completely specify the MGS protocol. In this section, we briefly describe the protocol engines. Since a full exposition of the protocol is beyond the scope of this paper, we refer the interested reader to [17] for more details.

The Local Client implements both TLB consistency and the client-side protocol for requesting page data. The Local Client runs on a processor each time it suffers a TLB fault. Three states in the Local Client correspond to the three states that a mapping can have in a processor's TLB: TLB_INV, TLB_READ, and TLB_WRITE. If the faulting processor finds a mapping in the local SMP, it copies the mapping and immediately transitions to the TLB_READ or TLB_WRITE state; otherwise, the page does not exist in the local SMP. In this case, the faulting processor enters the BUSY state and negotiates with the Server on the home SMP for replication of the page. Mutual exclusion within an SMP on page table state during TLB fault handling is achieved via a shared memory lock. There is one such lock for each mapping on each SMP.

The Remote Client performs page invalidation on a client SMP, and runs on the processor that owns the client-side copy of a page. When a request for page invalidation occurs, the Remote

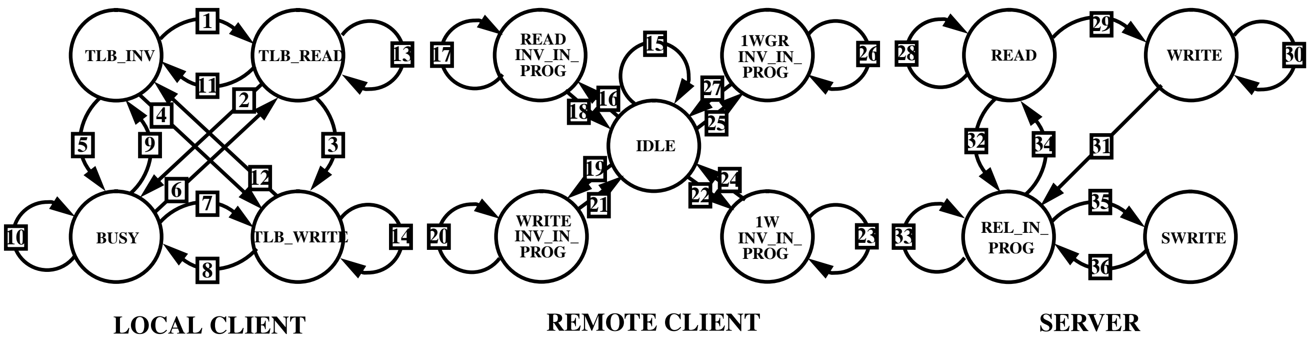

Figure 21: State transition diagram for the MGS Protocol.

Local Client ⇒ Remote Client Messages		Remote Client ⇒ Local Client Messages	
UPGRADE	Upgrade Local Page from Read to Write Privilege	UP_ACK	Acknowledge Upgrade
PINV_ACK	Acknowledge TLB Invalidation	PINV	Invalidate TLB Entry
P2INV_ACK	Acknowledge DUQ Invalidation	P2INV	Invalidate DUQ Entry
Local Client ⇒ Server Messages		**Server ⇒ Local Client Messages**	
RREQ	Read Data Request	RDAT	Read Data
WREQ	Write Data Request	WDAT	Write Data
REL	Release Request	RACK	Acknowledge Release
Remote Local Client ⇒ Server Messages		**Server ⇒ Remote Client Messages**	
ACK	Acknowledge Read Invalidate	INV	Invalidate Page
DIFF	Acknowledge Write Invalidate and Return Diff		
1WDATA	Acknowledge Single Writer Invalidate and Return Data	1WINV	Invalidate Single-Writer Page
WNOTIFY	Notify Upgrade from Read to Write Privilege		
ACK1W	Acknowledge Single Writer Status	1WGR	Grant Single-Writer Status

Table 5: Message types used to communicate between the Local Client, Remote Client, and Server machines in the MGS Protocol.

Arc	Event	Precondition	L	Side Effects	Out Message
1	RTLBFault	pagestate != INV	+/R	mapping → TLB, tlb_dir = $tlb\_dir \cup$ {src}	
2,5	WTLBFault	pagestate == READ	+/H	mapping → TLB, tlb_dir = $tlb\_dir \cup$ {src}	UPGRADE ⇒ l_home
3,4	WTLBFault	pagestate == WRITE	+/R	mapping → TLB, tlb_dir = $tlb\_dir \cup$ {src} DUQ = $DUQ \cup$ {addr}	
5	RTLBFault	pagestate == INV	+/H		RREQ ⇒ g_home
	WTLBFault	pagestate == INV	+/H		WREQ ⇒ g_home
6	RDAT		–/R	map page, tlb_dir = {src}, pagestate = READ	
7	WDAT		–/R	map page, tlb_dir = {src}, pagestate = WRITE DUQ = $DUQ \cup$ {addr}	
	UP_ACK		–/R	DUQ = $DUQ \cup$ {addr}	
8	Release		+/H	addr = DUQ–>$head$, DUQ = DUQ–>$tail$	REL ⇒ g_home(addr)
9	RACK	DUQ == ϕ	–/R		
10	RACK	DUQ != ϕ		addr = DUQ–>$head$, DUQ = DUQ–>$tail$	REL ⇒ g_home(addr)
11	PINV			invalidate TLB	PINV_ACK ⇒ l_home
12	PINV			invalidate TLB, DUQ = DUQ – {addr}	PINV_ACK ⇒ l_home
13,14	P2INV			DUQ = DUQ – {addr}	P2INV_ACK ⇒ l_home
15	UPGRADE			make twin, pagestate = WRITE	UP_ACK ⇒ src, WNOTIFY ⇒ g_home
16	INV	pagestate == READ	+/H	clean page, free page, count = $\lvert tlb\_dir \rvert$	PINV ⇒ tlb_dir
19	INV	pagestate == WRITE	+/H	make diff, free page, count = $\lvert tlb\_dir \rvert$	PINV ⇒ tlb_dir
22	1WINV		+/H	clean page, count = $\lvert tlb\_dir \rvert$	PINV ⇒ tlb_dir
25	1WGR		+/H	count = $\lvert tlb\_dir \rvert$	P2INV ⇒ 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 = ϕ, pagestate = INV	ACK ⇒ g_home
21	PINV_ACK	count == 0	–/R	tlb_dir = ϕ, pagestate = INV	DIFF ⇒ g_home
24	PINV_ACK	count == 0	–/R	tlb_dir = ϕ	1WDATA ⇒ g_home
27	P2INV_ACK	count == 0	–/R		ACK1W ⇒ g_home
28,30	RREQ			$read\_dir$ = $read\_dir \cup$ {src}	RDAT ⇒ src
29,30	WREQ			$write\_dir$ = $write\_dir \cup$ {src}	WDAT ⇒ src
29	WNOTIFY			$read\_dir$ = $read\_dir$ – {src}, $write\_dir$ = $write\_dir \cup$ {src}	
31	REL	$\lvert write\_dir \rvert$!= 1		count = $\lvert read\_dir \cup write\_dir \rvert$, rl = {src}, rd = wr = ϕ	INV ⇒ $read\_dir \cup write\_dir$
	REL	$\lvert write\_dir \rvert$ == 1, $\lvert read\_dir \rvert$!= 0		count = $\lvert read\_dir \cup write\_dir \rvert$, rl = {src}, rd = wr = ϕ	INV ⇒ $read\_dir$, 1WINV ⇒ $write\_dir$
	REL	$\lvert write\_dir \rvert$ == 1, $\lvert read\_dir \rvert$ == 0		count = 1, rl = {src}, rd = wr = ϕ	1WGR ⇒ $write\_dir$
32	REL			count = $\lvert read\_dir \cup write\_dir \rvert$, rl = {src}, rd = wr = ϕ	INV ⇒ $read\_dir$
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 = $rd \cup$ {src}	
	WREQ			wr = $wr \cup$ {src}	
	REL			rl = $rl \cup$ {src}	
	WNOTIFY				
	ACK1W	$\lvert rd \cup wr \rvert$!= 0		count = 1	INV ⇒ $write\_dir$
34	ACK	count == 0		merge diffs, $read\_dir$ = $write\_dir$ = ϕ	RACK ⇒ rl, RDAT ⇒ rd, WDAT ⇒ wr
	DIFF	count == 0		merge diffs, $read\_dir$ = $write\_dir$ = ϕ	RACK ⇒ rl, RDAT ⇒ rd, WDAT ⇒ wr
	1WDATA	count == 0		$read\_dir$ = $write\_dir$ = ϕ	RACK ⇒ rl, RDAT ⇒ rd, WDAT ⇒ wr
35	ACK1W	$\lvert rd \cup wr \rvert$ == 0			RACK ⇒ rl
36	RREQ			count = 1, rd = {src}, rl = wr = ϕ	INV ⇒ $write\_dir$
	WREQ			count = 1, wr = {src}, rl = rd = ϕ	INV ⇒ $write\_dir$

Figure 22: State transition table for the MGS Protocol. Italicized identifiers represent sets of processor IDs. <message>⇒<pid> denotes that we send <message> to <pid>. <message>⇒ <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 source processor ID of the current message.

Client invalidates the physical page, and sends TLB invalidation requests to all processors that have mapped the page. One of the INV_IN_PROG states is entered, depending on the access privilege and state of the local copy of the page, to wait for the TLB invalidations to complete. The Remote Client also performs page upgrade operations. A page upgrade happens when a processor tries to write to a page for which the local SMP only has read privilege. The Remote Client makes a twin of the read page, upgrades the privilege from read to write, and notifies the home SMP of the upgrade.

Finally, the Server handles the server-side protocol for page replication and release operations, and runs on the processor whose memory is home for the page. The Server has four states: READ, WRITE, SWRITE, and REL_IN_PROG. The READ state indicates that only read copies of the page are in the system, the WRITE state indicates the presence of multiple read-write copies, and the SWRITE state indicates a single write copy. The SWRITE state tracks pages in the special single-writer mode, as discussed in Section 3.2.3. The REL_IN_PROG state is entered when a release operation is invoked. All requests for replication that arrive when a page is in the REL_IN_PROG state are queued and then satisfied after the release completes.

Figure 22 gives the annotations for the transition arcs in Figure 21. Most of the notation is given in the caption of Table 22. The column labeled "L" is part of the state transition precondition and refers to the shared memory lock necessary for mutual exclusion on page table state at the client SMP. A "+" indicates that the lock must be acquired before the precondition is satisfied; otherwise, a "–" appears indicating that no lock acquire is necessary. A second value indicates the state of the lock after the state transition completes. The lock is either released or held, denoted by "R" and "H," respectively. The messages used to communicate between the three protocol engines that appear in Figure 22 are listed with brief descriptions in Table 5.