MGS: A Multigrain Shared Memory System

Donald Yeung, John Kubiatowicz, and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Parallel workstations, each comprising 10-100 processors, promise cost-effective general-purpose 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 Scalable Shared-memory Multiprocessors (DSSMPs).

This paper introduces the design of a shared memory system that uses multiple granularities of sharing, and presents an implementation on the Alewife multiprocessor, called MGS. Multigrain shared memory enables the collaboration of hardware and software shared memory, and is effective at exploiting a form of locality called *multigrain locality*. The system provides efficient support for fine-grain cache-line sharing, and resorts to coarse-grain page-level sharing only when locality is violated. A framework for characterizing application performance on DSSMPs is also introduced.

Using MGS, an in-depth study of several shared memory applications is conducted to understand the behavior of DSSMPs. We find that unmodified shared memory applications can exploit multigrain sharing. Keeping the number of processors fixed, applications execute up to 85% faster when each DSSMP node is a multiprocessor as opposed to a uniprocessor. We also show that tightly-coupled multiprocessors hold a significant performance advantage over DSSMPs on unmodified applications. However, a best-effort implementation of a kernel from one of the applications allows a DSSMP to almost match the performance of a tightly-coupled multiprocessor.

1 Introduction

Large-scale shared memory multiprocessors have traditionally been built using custom communication interfaces, high performance VLSI networks, and special-purpose hardware support for shared memory. These systems achieve good performance on a wide range of applications; however, they are costly. Despite attempts to make cost (in addition to performance) scalable, fundamental obstacles prevent large tightly-coupled systems from being cost effective. Power distribution, clock distribution, cooling, and other packaging considerations 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 high volume smaller-scale machines enjoy.

Appears in Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996.

In response to the high cost of traditional multiprocessors, many researchers have proposed building large-scale multiprocessors from commodity uniprocessor workstations that communicate across commodity networks. A "building wide" machine is cost effective because the components are high volume items and because specialized tightly-coupled packaging is not required. Achieving good performance across a wide range of applications, however, is difficult on these systems. While communication interfaces for commodity workstations have made impressive improvements, the best reported inter-workstation latency numbers are still an order of magnitude higher than for tightly-coupled machines [1]. High latency limits the granularity of sharing that can be effectively supported by a shared memory implementation that runs on uniprocessor workstations.

We believe that an important class of systems is quickly emerging that enables another way to build large-scale multiprocessors. This class of systems is the parallel workstation. A parallel workstation is any small- to medium-scale multiprocessor. A familiar example is the bus-based Symmetric Multiprocessor (SMP). Another example is the small- to medium-scale NUMA multiprocessor. The latter architecturally resembles large-scale tightly-coupled machines, but is targeted for smaller systems. We call these machines Scalable Shared-memory Multiprocessors (SSMPs). While all the issues in this paper apply equally to SMPs, we focus on SSMPs because they scale to larger configurations.

SSMPs are an attractive building block for large-scale multiprocessors. Because they are more affordable than their large-scale tightly-coupled ancestors, they will enjoy higher demand, thereby benefiting from the lower cost of high volume production. Furthermore, SSMPs have 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 which pay the cost of inter-workstation communication on every access to a remote memory module. We call a large-scale system built from SSMPs a Distributed Scalable Shared-memory Multiprocessor (DSSMP). Although the DSSMP concept has received attention in recent literature [2], the design issues remain unexplored, no working system has been developed, and no analysis of how applications behave on real DSSMP systems has been provided.

This paper proposes a shared memory system for DSSMPs. Our work introduces a complete shared memory protocol that allows hardware and software shared memory to collaborate in a way that is transparent to the programmer. The feasibility and correctness of our design is demonstrated in a working system that runs on the MIT Alewife platform [3]. We call this system MGS.

Our work also provides a framework for reasoning about how applications behave on DSSMPs. We define a form of locality, called *multigrain locality*, that multigrain shared memory systems can efficiently exploit. Applications that demonstrate multigrain locality can achieve good performance on DSSMPs. Furthermore, we define three metrics that characterize application performance on DSSMPs: the *breakup penalty*, the *multigrain potential*, and the *multigrain curvature*.

Using this framework, we conduct an in-depth study of five shared memory applications on the MGS system. We show that unmodified shared memory applications can exploit multigrain sharing. Keeping the total number of processors fixed, applications execute up to 85% faster when each DSSMP node is a multiprocessor as opposed to a uniprocessor. We also show that tightly-coupled multiprocessors still hold a significant performance advantage over DSSMPs on unmodified applications. However, a best-effort implementation of a kernel from one of our applications allows DSSMP performance to almost match the performance of a tightly-coupled multiprocessor.

Section 2 motivates the need for multigrain shared memory on DSSMPs, and introduces the notion of multigrain locality. It also presents our framework for reasoning about application performance on DSSMP systems. Section 3 presents a platform-independent implementation of MGS, and Section 4 discusses some implementation issues related to the Alewife platform. Section 5 presents our results on the MGS prototype. Section 6 discusses related work, and finally, Section 7 presents our conclusions.

2 Multigrain Shared Memory

This section describes what we mean by a DSSMP, and motivates the need for multigrain shared memory on DSSMPs. Then, the notion of *multigrain locality* is introduced. Finally, a framework for reasoning about application behavior on DSSMP systems is presented.

2.1 DSSMPs

Figure 1 shows a group of SSMPs that are interconnected by an external network. We call such a system a Distributed SSMP (DSSMP). DSSMPs have two types of networks that form the communication substrate: an internal network that connects processors within each SSMP, and an external network that connects the SSMPs. Each SSMP has efficient support for shared memory. Larger shared memory systems can be built by extending the shared memory mechanisms in each SSMP to other SSMPs through the external network. Shared memory at the inter-SSMP level is synthesized in software using the external network for messaging.

The network hierarchy in DSSMPs is matched by a hierarchy of shared memory latencies. A shared memory access that is satisfied within a single SSMP uses efficient hardware mechanisms for shared memory, and is, therefore, inexpensive. A shared memory access that spans SSMP boundaries needs to invoke software shared memory, and is, therefore, expensive. Moreover, inter-SSMP communication uses the external network, likely to be a commodity local area network that is both unreliable and untrusted. Providing reliability and security on these networks requires software protocol stacks that contribute additional overhead.

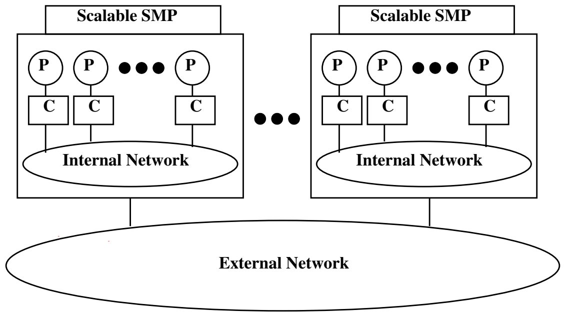


Figure 1: Building large-scale multiprocessors using SSMPs as the basic building block, and an external network for inter-SSMP communication.

2.2 Grain

In a shared memory system, grain refers to the unit of coherence. Choosing a grain size can significantly impact performance. A larger coherence grain amortizes the coherence overhead on a region of memory over more data, thus potentially reducing the overhead of shared memory actions. However, false sharing limits the effectiveness of larger grains. As grain becomes large, shared memory accesses to different parts of a memory region can interfere with one another and cause coherence actions to occur even though no true dependences exist between the accesses.

The optimal grain size depends on both the cost of maintaining coherence on each memory region, and the sharing behavior of applications. In DSSMPs, the cost of a shared memory access can vary significantly depending on whether the access is handled in hardware or in software. This suggests the use of two separate grains: a cache-line sized grain when using hardware shared memory, and a larger page-sized grain when using software shared memory.

Other solutions to the grain size problem have explored variable grains [4, 5]. These solutions leverage application-specific knowledge to select the optimal grain. While this can be effective, it is difficult for a compiler and painful for a programmer. Multigrain shared memory does not rely on application-specific knowledge. Instead, it supports fine-grain sharing when access patterns exhibit locality, and resorts to coarse-grain sharing only when locality is violated.

2.3 Locality

DSSMPs exploit cluster locality and multigrain locality, each of which can be informally defined based on the notion of working sets.

The working set property [6] exhibited by a processor's memory references is a statement about locality of reference. The working set property states that the size of the set of unique memory blocks

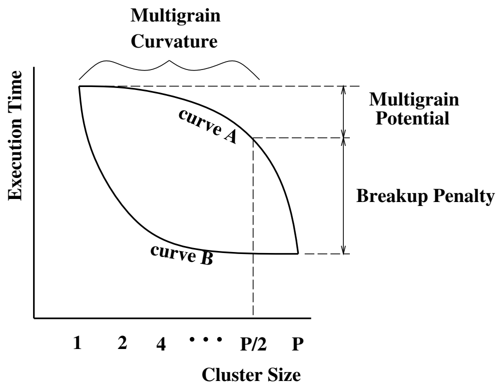


Figure 2: Two hypothetical curves that plot execution time as a function of cluster size. Breakup penalty, multigrain potential, and multigrain curvature are labeled for curve A.

accessed by a processor in a given interval of time is small compared to the total number of memory blocks in the program's dataset.

Cluster locality, as defined in [7], states that the working sets of processors in the same cluster are more likely to intersect than those of processors in different clusters. Cluster locality can be exploited by any hierarchical shared memory system.

In addition to cluster locality, we introduce multigrain locality. We define multigrain locality in the following way.

Multigrain Locality. Let the union of the individual processor working sets in a cluster be the cluster working set. Multigrain locality states that the sharing between processors in different clusters resulting from the intersection of cluster working sets uses a coarser grain than the sharing between processors in the same cluster resulting from the intersection of processor working sets.

Multigrain locality is an extension of cluster locality. The existence of multigrain locality implies the existence of cluster locality, but the converse is not true. While cluster locality can be exploited by any hierarchical shared memory system, multigrain locality can only be exploited by hierarchical systems that use different grains of sharing at different levels in the hierarchy. The implication of multigrain locality on applications is that *some* fine-grain sharing is supported.

2.4 DSSMP Performance Framework

Two key system parameters describe a DSSMP configuration: the total number of processors, P, and the number of processors per SSMP or cluster size, C. In our framework, we keep P fixed and vary C from 1 to P. The endpoints of this range are interesting because each represents a collapse of the DSSMP network hierarchy. At $C = 1$, each SSMP is a uniprocessor, so there is no internal network. This means that all shared memory accesses to remote locations use software and share at page granularity. Conversely, at $C = P$, there is only 1 SSMP, and it is the entire system. There is no external network. All remote accesses are handled in hardware and share at cache-line granularity. Larger values of C correspond to DSSMPs that rely more on hardware for its shared memory implementation.

Figure 2 shows two hypothetical curves that plot performance on a DSSMP. Execution time is plotted against the cluster size parameter, C, in powers of 2 for a total system size, P. We compare the time at cluster size $C = P$ against the times at cluster sizes $C = 1 \ldots \frac{P}{2}$. This ratio is the slowdown suffered by breaking a tightly-coupled machine of P processors into N clusters of size $\frac{P}{N}$ processors each, for a range of N.

We identify three metrics that characterize an application's behavior on DSSMP systems. The curve named "curve A" in Figure 2 has been labeled with these metrics.

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

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

Multigrain Curvature. The shape of the curve across the multigrain potential is the "multigrain curvature." A concave curvature indicates most of the multigrain potential is achieved at large cluster sizes, while a convex curvature indicates most of the multigrain potential is achieved at small cluster sizes.

Curve A in Figure 2 represents an application that has a high breakup penalty. The multigrain potential is small, and the multigrain curvature is concave. Curve A type applications are not well suited for DSSMPs. In contrast, Curve B has a very small breakup penalty indicating essentially no loss in introducing some software in the shared memory implementation. It has a large multigrain potential indicating good benefits derived from capturing fine-grain sharing in clusters, and the multigrain curvature is convex with a steep slope at small cluster sizes. This indicates that most of the multigrain potential is achieved using small cluster sizes. The implication for Curve B type applications is that they will perform well on DSSMPs constructed from small-scale multiprocessors.

3 MGS System Design

This section discusses the design of the MGS system including the multigrain shared memory protocol we developed, and the support for synchronization. The discussion defers platform-specific implementation details of the system to Section 4.

3.1 MGS Shared Memory

MGS supports replication of data at both page and cache-line granularities. Between SSMPs, coherence actions occur at the granularity of a page. Once a page is resident in the memory of an SSMP, processors within the SSMP 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. SSMPs other

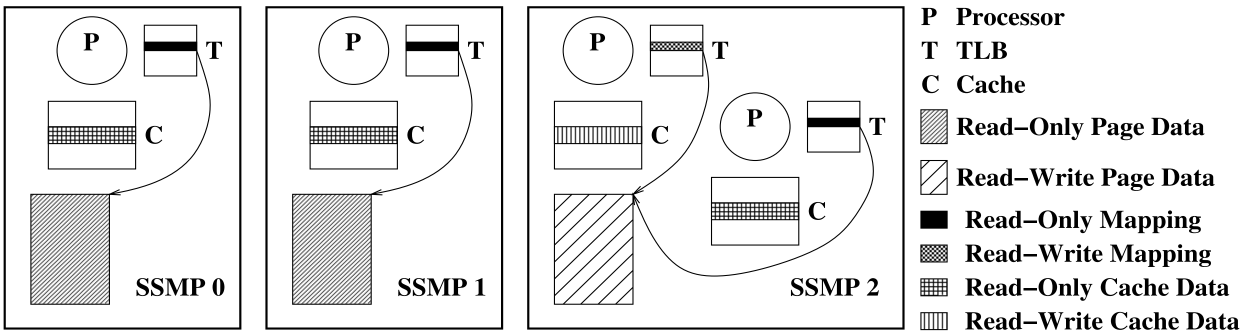


Figure 3: Distribution of a page of data across three SSMPs in the MGS system. Only the processors involved in sharing are shown; there are other processors in each SSMP that are not shown.

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 SSMP has a *physical local copy*, processors in the SSMP 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 3 illustrates how data from a single page gets distributed to SSMPs and processors. SSMP0 contains the physical home copy which is itself in read mode. A single processor in SSMP0 has a read-only mapping and has read some of the data. SSMP1 and SSMP2 have read-only and read-write physical local copies, respectively. Since SSMP1 has a read-only copy, its processors can only map it in read-only mode. Processors in SSMP2, 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.

3.1.1 Memory Consistency

The page-based software protocol in MGS is release consistent [1], invalidation-based, and supports multiple writers. The specifics of the page-based protocol borrow heavily from Munin [8]. Like Munin, MGS uses a delayed update queue (DUQ) 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 SSMPs. Eager invalidation was chosen for implementation simplicity.

In addition to the basic Munin techniques, MGS employs a novel optimization, called the *single-writer optimization*. Twins are made at request time for read-write pages as in Munin. At invalidation time, if there is only one write copy outstanding, the entire page is sent to the home instead of computing a diff, and the read-write copy is allowed to remain cached. This optimization presents two benefits. First, diff computation overhead is traded off for higher communication bandwidth to send the entire page; for our implementation of MGS, this is a worthwhile tradeoff. And second, leaving a cached copy with the writer after a release rewards sharing within the same SSMP across release points at the expense of slowing down data migration between SSMPs. This policy gives preference to multigrain sharing.

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

3.1.2 Protocol Engines

Three software protocol engines implement the MGS Protocol: the Local Client, the Remote Client, and the Server. Figure 4 shows the state transition diagram for these protocol engines.

The Local Client maintains consistency on mapping state, and implements the client-side protocol for requesting page data. The Local Client runs on the processor that 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 SSMP, it copies the mapping and immediately transitions to the TLB_READ or TLB_WRITE state; otherwise, the page does not exist in the local SSMP. In this case, the faulting processor enters the BUSY state and negotiates with the Server on the home SSMP for replication of the page. We use a first-touch placement policy for replicated pages; once placed, pages do not migrate within the SSMP. Mutual exclusion within an SSMP 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 SSMP.

The Remote Client performs page invalidation on the client side, and runs on the processor that owns the client-side copy of a page. When a request for page invalidation occurs, the Remote Client invalidates the physical page, and sends TLB invalidation requests to all processors that have mapped the page. The INV_IN_PROG state is entered 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 SSMP 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 SSMP 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 three states: READ, WRITE, and REL_IN_PROG. The READ state indicates that only read copies of the page are in the system, while the WRITE

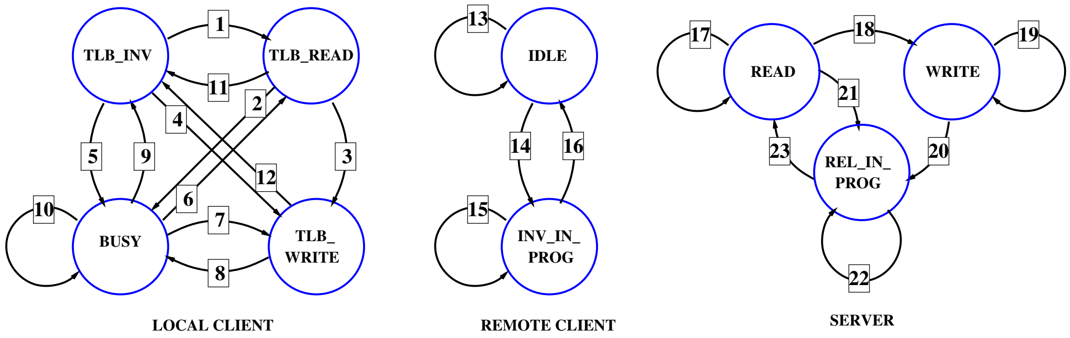


Figure 4: State transition diagram for the MGS Protocol.

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	$DUQ == \phi$	–/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	UPGRADE			make twin, pagestate = WRITE	UP_ACK ⇒ src, WNOTIFY ⇒ g_home
14	INV	pagestate == READ	+/H	clean page, free page, count = $\vert tlb\_dir \vert$, tt = 1	PINV ⇒ tlb_dir
	INV	pagestate == WRITE	+/H	make diff, free page, count = $\vert tlb\_dir \vert$, tt = 2	PINV ⇒ tlb_dir
	1WINV		+/H	clean page, count = $\vert tlb\_dir \vert$, tt = 3	PINV ⇒ tlb_dir
15	PINV_ACK	count != 0		count = count – 1	
16	PINV_ACK	count == 0, tt == 1	–/R	$tlb\_dir = \phi$, pagestate = INV	ACK ⇒ g_home
	PINV_ACK	count == 0, tt == 2	–/R	$tlb\_dir = \phi$, pagestate = INV	DIFF ⇒ g_home
	PINV_ACK	count == 0, tt == 3	–/R	$tlb\_dir = \phi$	1WDATA ⇒ g_home
17,19	RREQ			$read\_dir = read\_dir \cup \{src\}$	RDAT ⇒ src
18,19	WREQ			$write\_dir = write\_dir \cup \{src\}$	WDAT ⇒ src
18	WNOTIFY			$read\_dir = read\_dir - \{src\}$,	
				$write\_dir = write\_dir \cup \{src\}$	
20	REL	$\vert write\_dir \vert$!= 1		count = $\vert read\_dir \cup write\_dir \vert$,	INV ⇒ $read\_dir \cup write\_dir$
				$rl = \{src\}, rd = wr = \phi$	
	REL	$\vert write\_dir \vert == 1$		count = $\vert read\_dir \cup write\_dir \vert$,	INV ⇒ $read\_dir$, 1WINV ⇒ $write\_dir$
				$rl = \{src\}, rd = wr = \phi$	
21	REL			count = $\vert read\_dir \cup write\_dir \vert$,	INV ⇒ $read\_dir$
				$rl = \{src\}, rd = wr = \phi$	
22	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				
23	ACK	count == 0		merge diffs, $read\_dir = write\_dir = \phi$	RACK ⇒ rl, RDAT ⇒ rd, WDAT ⇒ wr
	DIFF	count == 0		merge diffs, $read\_dir = write\_dir = \phi$	RACK ⇒ rl, RDAT ⇒ rd, WDAT ⇒ wr
	1WDATA	count == 0		$read\_dir = write\_dir = \phi$	RACK ⇒ rl, RDAT ⇒ rd, WDAT ⇒ wr

Table 1: 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.

Local Client ⇒ Remote Client Messages		Remote Client ⇒ Local Client Messages	
UPGRADE	Upgrade Local Page from Read to Write Privilege	PINV	Invalidate TLB Entry
PINV_ACK	Acknowledge TLB Invalidation	UP_ACK	Acknowledge Upgrade
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		

Table 2: Message types used to communicate between the Local Client, Remote Client, and Server engines in the MGS Protocol.

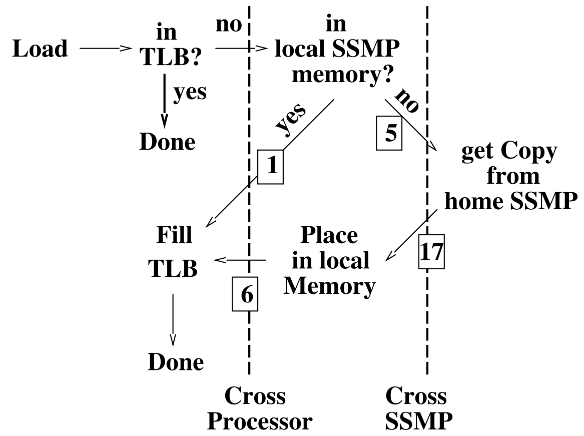


Figure 5: A load operation on MGS. Labels correspond to state transitions in the state diagram shown in Figure 4.

state indicates the presence of at least one read-write copy. 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.

Table 1 gives the annotations for the transition arcs in Figure 4, and is a complete specification for the MGS Protocol. Most of the notation is given in the caption of Table 1. 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. A "+" indicates that the lock must be acquired via spin-waiting[2] 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 are described in Table 2.

To illustrate how the different parts of the MGS protocol interact, Figure 5 shows a load operation. When a processor performs a load, it first checks its local TLB for a mapping. If a mapping is found in the TLB, a translation produces a physical address, and the data is provided to the processor through the hardware cache coherence mechanism on the SSMP. If, however, a TLB fault occurs, the Local Client is invoked and tries to find a mapping on the local SSMP. If the SSMP has a local physical copy of the desired page, the mapping will be found and the TLB filled. This corresponds to state transition 1 in Figure 4. If a local physical page cannot be found, the Local Client transitions to the BUSY state (transition 5) and sends an RREQ message to the home SSMP to request a read copy of the page. On the home SSMP, the Server protocol is invoked to send an RDAT message back to the client SSMP (transition 17). When this data arrives at the client SSMP, the Local Client transitions to READ state (transition 6), and the mapping is filled in the original processor's TLB.

[2] In actuality, only tasks spin-wait on locks. Handlers that acquire locks test the lock, and queue if the lock is busy in order to avoid deadlock.

3.2 MGS Synchronization

The MGS system is accompanied by a user-level synchronization library that is cognizant of the hierarchy inherent in the MGS system. The goal of the synchronization primitives is to contain communication within an SSMP whenever possible. In this section, we discuss the techniques used for barriers and locks; both of these have been proposed by Cox et al [2].

The MGS barrier is a tree barrier that is structured to match the hierarchical structure of the DSSMP. The first level in the tree synchronizes all processors on the same SSMP. The second level synchronizes all the SSMPs. This achieves a minimum inter-SSMP message count of two messages per SSMP (one for the combine, and one for the release).

The MGS lock is a token-based distributed lock. Each MGS lock consists of a local lock on each SSMP, and a single global lock. A token is passed among the local locks; obtaining ownership of the token requires acquiring the global lock. Acquires to the local lock succeed if the local lock owns the token. Once a local lock owns a token, repeated acquires from the same SSMP succeed without inter-SSMP communication. Communication between SSMPs occurs only when consecutive acquires from different SSMPs necessitates acquiring the global lock.

4 MGS on Alewife

In this section, we discuss the implementation of MGS on the Alewife platform. We give a brief overview of Alewife, and then discuss four concerns for supporting MGS on Alewife: address translation, inter-SSMP communication, use of active messages, and global coherence.

4.1 The Alewife Multiprocessor

Alewife is a distributed memory multiprocessor that has hardware support for the shared memory abstraction. 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 messaging interface with DMA capability [9]. DMA data in messages are locally coherent.

4.2 Support for MGS

4.2.1 Software Virtual Memory

MGS relies on virtual memory. On machines with hardware support for virtual memory, MGS would require special TLB fault handlers. Because Alewife does not support virtual memory, MGS performs address translation in software. The compiler identifies which memory accesses need translation and emits code in-line prior to these accesses to handle translation. The in-lined code obtains 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.

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 variables, are assumed to be unmapped and incur no translation overhead. Translation for pointer dereferences is slightly more expensive than translation for distributed arrays. This is because all distributed arrays are mapped objects whereas pointers can point to both mapped and unmapped objects. The extra overhead in translating pointer dereferences is spent determining whether the pointer address is virtual or physical; this is possible because the virtual and physical spaces have disjoint address assignments.

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. Requests to invalidate a mapping interrupt the processor owning 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 was designed to be reentrant).

4.2.2 Simulating LANs

Alewife is a tightly-coupled multiprocessor. We emulate a DSSMP on Alewife by logically partitioning the Alewife nodes into SSMPs. The entire machine is still tightly coupled; however, we only allow processors to map pages that reside in their local logical SSMP. An access to a page that resides outside the local SSMP causes a fault and subsequent invocation of the MGS Protocol.

Communication across logical SSMPs required for the MGS protocol uses the fast Alewife messaging mechanisms. To better model the cost of inter-SSMP communication, all messages between logical SSMPs are queued at the sending processor and a timer interrupt is set for some amount of delay. When the timer interrupt occurs after the delay has expired, the message is taken off the queue and actually sent. This technique models the inter-SSMP communication cost as a fixed latency. Our implementation of MGS does not account for contention in the LAN, nor in the 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.

4.2.4 Global Coherence

MGS uses messages with DMA to efficiently transport page-size data between SSMPs and further requires that data to be globally coherent with respect to the SSMP. On Alewife, DMA in messages is only locally coherent; global coherence [9] is synthesized in software through a process called *page cleaning*.

Cleaning a page requires generating invalidations for every cache line in the page. A reasonably efficient way to accomplish this is to issue write prefetches some number of cache lines ahead in a tight loop, and at each iteration of the loop, to do a store instruction followed by a flush of the cache line. The write prefetch causes the cache line to be invalidated from all other caches. If initiated far enough in advance, the write prefetch will hide the latency of the invalidation and the store instruction will hit in the processor's cache. The store instruction is necessary because Alewife ignores prefetch requests when system resources are limited, or if the memory directory for that cache line is busy. The store instruction ensures that the invalidation actually happens in the event that the prefetch is ignored. After the flush completes, the cache line is guaranteed to be cleaned from the system.

One problem with this scheme is that the latency of invalidation for widely read shared data can be very high and hard to hide with prefetching. However, invalidation of read-only data can be removed from the critical path of page invalidation because there is no coherence issue with read-only data. While MGS does not currently make use of this observation, we are exploring an optimization in a future implementation of MGS that will.

5 Results

We first present measurements that show the cost of primitive MGS operations, and then we present extensive results showing the performance of five shared memory applications running under MGS.

Hardware Shared Memory	
Cache Miss Local	11
Cache Miss Remote	38
Cache Miss 2-party	42
Cache Miss 3-party	63
Remote Software	425
Software Virtual Memory	
Distributed Array Translation	18
Pointer Translation	24
Software Shared Memory	
TLB Fill	1037
Inter-SSMP Read Miss	6982
Inter-SSMP Write Miss	16331
Release (1 writer)	14226
Release (2 writers)	32570

Table 3: Shared Memory Costs on MGS.

Application	Problem Size	Seq	S32
Jacobi	1024×1024 Grid, 10 Iterations	1618	30.0
Matrix Multiply	256×256 Matrices	3081	26.9
TSP	10-City Tour	54.2	23.0
Water	343 Molecules, 2 Iterations	1993	26.9
Barnes-Hut	2K Bodies, 3 Iterations	977	13.8
Water-kernel	512 Molecules, 1 Iteration	1540	26.7

Table 4: Applications and their problem sizes. The column labeled "Seq" reports sequential running time in millions of cycles, and the column labeled "S32" reports the speedup observed on 32 processors.

5.1 Micro Measurements

Table 3 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 top group measures the cost of the Alewife hardware shared memory. These latencies represent the penalty for various types of cache misses. They do not include the overhead of address translation. The entry labeled "Remote Software" reports the cost of a read miss to a cache line under software directory control. All measurements are taken for load misses; store misses take slightly longer, and can be found in [3].

The second group of measurements shows the cost of software translation. Translation for both distributed array objects and general pointers are shown. Finally, the bottom group of measurements report on the cost of MGS' software coherence protocol. All measurements were taken assuming a 1K-byte page size and a 0 cycle delay for communication between SSMPs.

5.2 Application Performance

In this section, we report on the performance of five shared memory applications running on the MGS system: Jacobi, Matrix Multiply, the Traveling Salesman Problem, Water, and Barnes-Hut. Jacobi is a 2-D grid relaxation program. Matrix Multiply multiplies two square matrices. TSP computes the solution to a 10-city traveling salesman problem using a branch and bound algorithm, and a centralized work queue to distribute work. Water is a benchmark from the original SPLASH suite [10]. It is a 3-D simulation of the motion of water molecules. Barnes-Hut is also taken from the original SPLASH suite. It is a 3-D hierarchical n-body simulation. A modification was made in the way cells are allocated to relieve severe contention on a centralized lock. The modification is similar to a modification that is available in the SPLASH-2 [11] version of this code.

Table 4 lists the five applications, including the Water-kernel, which is a special version of Water to be explained in Section 5.2.3. For each application, the problem size used, the sequential runtime, and the speedup achieved on 32 processors is shown. The sequential runtime includes the overhead for software virtual memory. The speedups reported are calculated from executions on 32 processors without MGS, but with software virtual memory. Since the overhead of software virtual memory is perfectly parallelizable, its inclusion tends to improve speedup. The speedups reported in Table 4 show that software virtual memory does not render all the applications embarrassingly parallel.

5.2.1 Runtime Breakdowns

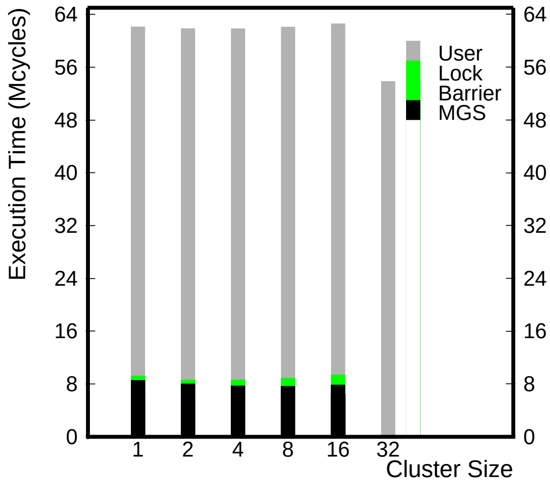
Figures 6 through 10 report the breakdown of runtimes for our applications. For all applications, we use a 1K-byte page size, and an inter-SSMP message delay of 1000 cycles. Each graph shows the execution time for a single application on a 32 processor Alewife machine running the MGS system. The different bars in each graph correspond to six different cluster sizes between 1 and 32, increasing by powers of 2. For the 32 processor data points, we substitute the normal MGS calls with null calls, and instead of linking with the MGS cluster-based synchronization library, we link with the P4 libraries. Therefore, the 32 processor data points show performance on a tightly-coupled multiprocessor without any MGS overheads, aside from software virtual memory.

All runtimes reported 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 software coherence, labeled User, Lock, Barrier, and MGS, respectively. The user component not only counts useful cycles in user code, but it also counts cycles spent in software address translation and hardware shared memory stall time. The synchronization components include both the overhead of executing synchronization code and waiting on synchronization conditions. Finally, the software coherence component represents all time spent running the MGS Protocol[3].

Figures 6 and 7 show the runtime breakdowns for Jacobi and Matrix Multiply. The breakup penalty, as defined in Section 2.4, is low for both these applications, 16% for Jacobi, and essentially 0% for Matrix Multiply. However, there is no performance gain in the multigrain region–the performance of these applications is independent of cluster size. Both applications have long computation phases that read and write large contiguous regions of data without data dependences. The coarse-grain nature of the sharing patterns allow these applications to run well regardless of the shared memory implementation.

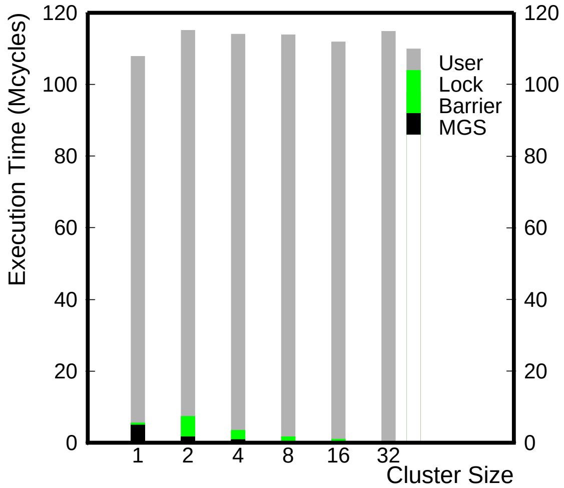
TSP performs quite differently. Two factors make performance on TSP poor. First, a centralized work queue that holds partial tours is a severe bottleneck. Processors require mutual exclusive access to the queue both when placing new work on the queue, and when removing work to perform. Second, false sharing is a factor as well.

[3]The 32 processor data points report only a user component. While the MGS component is truly zero for these runs, there is a non-zero synchronization component that has been folded into the user component because we did not instrument cycle counting in the P4 library.



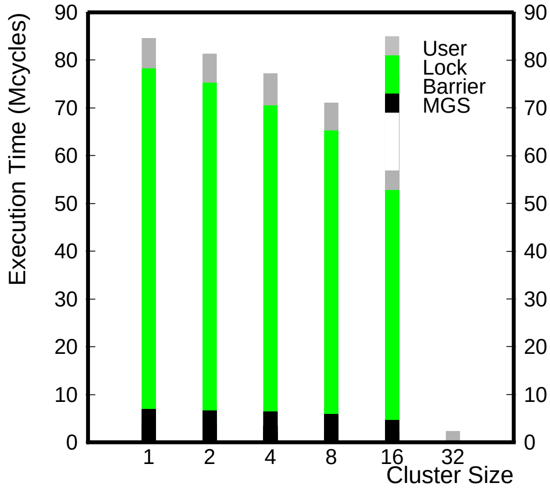
Multigrain Potential: 0%
Breakup Penalty: 16%

Figure 6: Runtime breakdown for Jacobi.



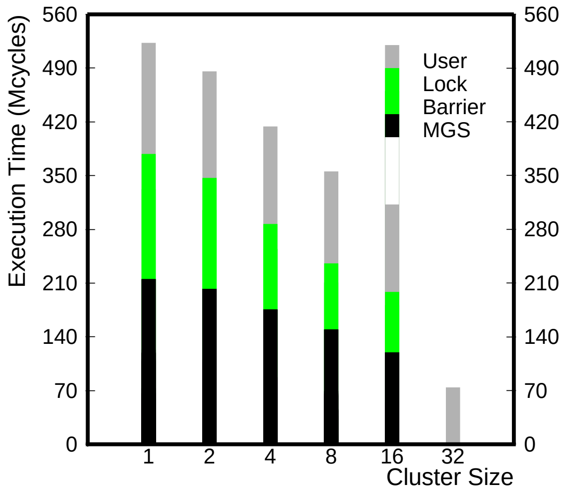
Multigrain Potential: <0%
Breakup Penalty: <0%

Figure 7: Runtime breakdown for Matrix Multiply.



Multigrain Potential: 49%
Breakup Penalty: 2270%

Figure 8: Runtime breakdown for TSP.



Multigrain Potential: 67%
Breakup Penalty: 322%

Figure 9: Runtime breakdown for Water.

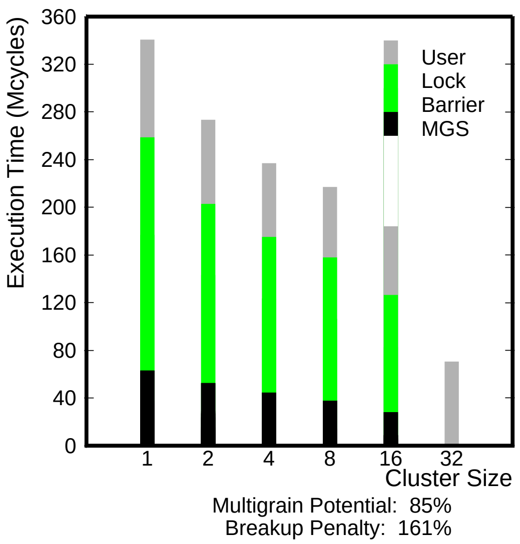


Figure 10: Runtime breakdown for Barnes-Hut.

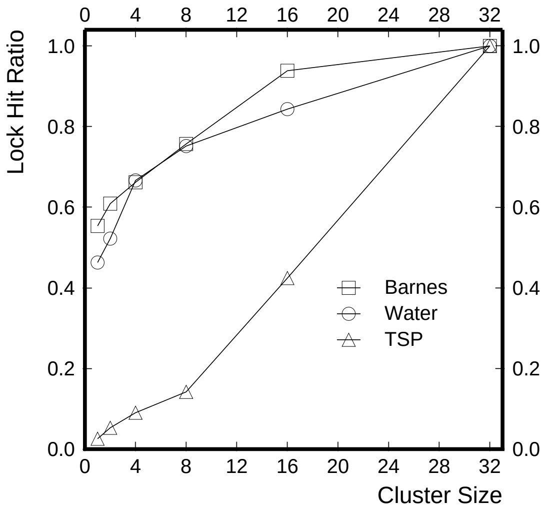


Figure 11: Hit Rate for MGS lock as a function of cluster size.

In TSP, the unit of data for computation is the path element which holds partially evaluated tours. Path elements for a 10-city tour are 56 bytes in size. Because they are contiguously allocated and then randomly assigned to processors from the work queue, there is large amounts of false sharing.

Figure 8 shows the runtime breakdown for TSP. The breakup penalty is quite large. TSP runs more than a factor 25 slower on a DSSMP than on a tightly-coupled machine. Although the multigrain potential is good, about 49%, the multigrain curvature is concave (*i.e.*, most of the multigrain potential is dropped across large cluster sizes). Extremely high lock overhead reflects the bottleneck at the centralized work queue. Although the critical section for work queue operations is very short, the overall lock overhead is high because of an effect we call *critical section dilation*. Under hardware cache coherence, the lock protecting the critical section is held for a short amount of time. But under software page coherence, a consistency operation in software happens before the lock is released. This is a long latency operation and thus significantly increases the critical section length.

Figure 9 shows the runtime breakdown for Water. Water exhibits a somewhat better breakup penalty over TSP, 322%, and it shows a much higher multigrain potential of 67%. The performance gains are due to reductions in both lock overhead and software coherence overhead. The access patterns in Water encourage multigrain sharing. In Water, a global molecule array is distributed amongst processors. Each processor accesses this array linearly starting from the portion it owns. Adjacent portions are distributed to processors that are physically close. Processors residing in the same SSMP share the global molecule array at fine granularity and can avoid software coherence. Also, there is a lock associated with each molecule. Ownership of the lock tends to pass among processors in the same SSMP thus reducing lock latency.

A significant limitation to higher performance in Water is barrier overhead. This is due to load imbalance in software coherence processing. The number of molecules in our runs is not divisible by the number of processors, so the number of molecules that are home on a particular processor is not balanced across all processors. In addition, the processor that is home to the data structure that tracks global statistics receives more coherence traffic than the others. Lock overhead is fairly significant as well and is due to critical section dilation.

Finally, Figure 10 shows the runtime breakdown for Barnes-Hut. Again, we see a fairly high breakup penalty (though much better than Water) of 161%, but Barnes-Hut exhibits the highest multigrain potential yet, 85%. Barnes-Hut has convex multigrain curvature. Performance is surprisingly good considering Barnes-Hut is a much finer-grained application than Water. Lock overhead is high due to critical section dilation during a parallel tree build phase which constructs the main data structure at the beginning of each iteration. The frequency of software shared memory consistency operations in this phase of the computation is very high. Also, barrier overhead is very significant for the same reasons as in Water.

5.2.2 MGS Lock Results

In this section, we look at the performance of the MGS token-based lock. Our metric for lock performance is the lock hit ratio. Lock hit ratio is defined as the number of lock acquires that succeed without inter-SSMP communication divided by the total number of lock acquires. Figure 11 plots the lock hit ratio as a function of cluster size for our applications, excluding the embarrassingly parallel applications.

We find that the lock hit ratio increases monotonically with increasing cluster size for all the applications. Furthermore, hit ratio is higher for the applications that exploit multigrain sharing (*i.e.,* Water and Barnes-Hut have a better lock hit rate than TSP, especially at small cluster sizes).

5.2.3 Enhancing Multigrain Locality in Water

Previous sections examine the performance of unmodified shared memory programs on DSSMPs. In this section, we explore in-

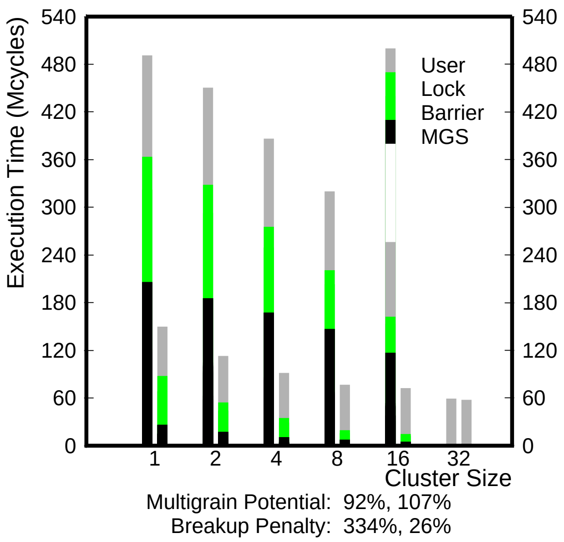


Figure 12: Runtime breakdown for Water-kernel without (left set of bars) and with (right set of bars) loop transformation.

creases in performance that can be achieved through a best-effort implementation which improves multigrain locality.

We consider the force interaction kernel from the Water application. The kernel accounts for most of the application's execution time. It consists of a doubly nested loop that traverses the main molecule array to perform the N-squared force interactions. Each iteration through the loop performs a pair-wise interaction and writes both molecules. Because of read sharing on the molecules, these writes cause invalidation traffic.

A loop transformation was performed, by hand, to improve the kernel's multigrain locality. In this loop transformation, the main molecule array is tiled such that there are two tiles per SSMP, and the computation of interactions proceeds in phases. In each phase, an SSMP chooses two tiles and the processors in the SSMP perform all possible pair-wise interactions between molecules from the two tiles. A schedule for tile assignments to SSMPs is computed such that in any given phase, each tile is assigned to exactly one SSMP so that the SSMP has exclusive access to the tile.

Figure 12 shows the performance of both the unoptimized and optimized kernels. The unoptimized kernel performs much like the original Water application; however, the optimized kernel shows a large performance improvement. Most notably, the breakup penalty drops from 334% to 26%. Furthermore, a large multigrain potential is still observable, 107%, and the multigrain curvature is convex. The optimized kernel has perfect multigrain locality. Within each phase, all sharing occurs within the SSMPs, and relies on hardware cache coherence. Only across phases does software page-grain communication occur. As cluster size increases, performance improves because the amount of work in each tile grows thus reducing the frequency of page-based coherence at phase boundaries.

6 Related Work

Several researchers have investigated the potential for clustering in shared memory systems. Cox et al [2] study a system built from 8-way bus-based multiprocessors connected over an ATM network. Three key differences distinguish our work from the Cox paper. First, the Cox paper is a simulation study, whereas our work presents a full implementation. The DSM system in the Cox study makes many simplifying assumptions which are dealt with directly in our system. Second, we identify the ability to leverage multiple grains of sharing as the key to achieving performance on DSSMP-like systems, and we provide a framework for reasoning about application performance on multigrain systems. Lastly, the Cox study uses bus-based multiprocessors. Our study provides an implementation for NUMA machines.

Other systems exploit clustering at a level closer to the processor, typically in the first or second level cache. These systems include VMP-MC [12], DASH [13], and KSR [14]. The benefit of clustering on these systems has been studied [15]. Interference misses due to limited cache capacity and associativity can reduce the benefits of clustering close to the processor. MGS clusters at the main memory level and thus does not suffer from these effects. Also, the use of MGS-style multiple grains is prohibitive in these systems because support for both intra- and inter-cluster accesses occurs in hardware; using a single cache-line grain is desirable for simplicity.

Several studies have explored multiple or variable grains of sharing. Galactica Net [16] allows both cache-line size and page-size grains of sharing; however, they rely on hardware support between clusters to implement update-based protocols at cache-line grain to efficiently support fine-grain write sharing between clusters. Chandra et al [17], Shared Regions [4], and CRL [5] allow coherence to happen at variable grains, but treat all processors equally (*i.e.,* no clustering), and require user annotations to identify the regions. MGS runs shared memory applications unmodified. The only requirement is that the programmer uses a release consistent memory model.

Finally, there is a large body of work on software distributed shared memory. Our particular DSM implementation borrows from the Munin system [8]. Other approaches have tried to minimize message traffic and optimize memory management policies [18, 19, 20, 21]. MGS would benefit from these techniques.

7 Conclusion

Using small- to medium-scale multiprocessors as the basic building block for large-scale multiprocessors is 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. Second, SSMPs already have hardware support for shared memory. Systems that can leverage the efficient hardware mechanisms in each SSMP have an advantage over conventional uniprocessor workstations.

MGS demonstrates how shared memory for DSSMPs can be designed. We propose a protocol that couples hardware cache coherence with software distributed shared memory. We identify the difficulty in maintaining consistency on data that has been distributed in a multigrain fashion, and discuss the problem of reclaiming such distributed data via page cleaning. The feasibility and correctness of our design is demonstrated by a working implementation that runs on the Alewife platform.

The key to good application performance on multigrain shared memory systems is the ability to exploit multigrain locality. Multigrain locality strikes a balance between the support of tightly-coupled multiprocessors and networks of workstations in that *some* fine-grain sharing can be supported efficiently. Our work quantifies the extent to which shared memory programs exhibit multigrain sharing. To enable such a study, we define a framework for reasoning about application performance on DSSMPs based on three metrics: breakup penalty, multigrain potential, and multigrain curvature.

Using our framework, we study five shared memory applications on the MGS system. Our first conclusion is that the multigrain potential is significant, ranging from 38% to 85%. This is encouraging because it indicates that applications can leverage multigrain sharing even without restructuring; therefore, using small-scale multiprocessors as software DSM nodes is better than using uniprocessor workstations. Another conclusion is that for unmodified applications, the breakup penalty is high. Excluding embarrassingly parallel applications, the breakup penalty ranges from 161% to 2270%. That is, fully tightly-coupled machines still offer a significant performance advantage over DSSMPs on unmodified applications. However, we found that a multigrain locality enhanced implementation of the force interaction kernel from the Water application was able to reduce the breakup penalty substantially. On our optimized kernel, the breakup penalty improved by dropping from 334% to only 26%, while a very large multigrain potential, 107%, was still visible. Shared memory allows a programmer to develop applications on a DSSMP quickly, and then improve incrementally the multigrain locality either manually or by using a compiler or runtime for increased performance. We are currently investigating compiler and runtime support for multigrain locality.

8 Acknowledgments

This research is funded in part by ARPA contract #N00014-94-1-0985, in part by NSF Experimental Systems grant #MIP-9504399, and in part by a NSF Presidential Young Investigator Award. The authors would like to thank Kavita Bala, Fred Chong, Fredrik Dahlgren, Matt Frank, Silvina Hanono, Kirk Johnson, Kathy Knobe, Victor Lee, and Deborah Wallach for providing valuable comments on early drafts of this paper.

References

[1] 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.

[2] 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, San Diego, California, 1994.

[3] 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.

[4] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Principles and Practices of Parallel Programming, 1993*, pages 229–238, San Diego, CA, May 1993.

[5] 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.

[6] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[7] Timothy Mark Pinkston and Sandra Johnson Baylor. Parallel Processor Memory Reference Analysis: Examining Locality and Clustering Potential. RC 15801, IBM T. J. Watson Research Center, May 1990.

[8] 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.

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

[10] 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.

[11] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[12] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Multi-Level Shared Caching Techniques for Scalability in VMP-MC. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 16–24, Jerusalem, Israel, June 1989.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[14] Kendall Square Research, Inc., 170 Tracer Lane, Waltham, MA 02154. Kendall Square Research Technical Summary, 1992.

[15] Andrew Erlichson, Basem A. Nayfeh, Jaswinder P. Singh, and Kunle Olukotun. The Benefits of Clustering in Shared Address Space Multiprocessors: An Applications-Driven Investigation. Technical Report CSL-TR-94-632, Computer Systems Laboratory, Stanford University, November 1994.

[16] Andrew W. Wilson Jr. and Richard P. LaRowe Jr. Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture. *Journal of Parallel and Distributed Computing*, 15(4):351–367, 1992.

[17] Rohit Chandra, Kourosh Gharachorloo, Vijayaraghavan Soundararajan, and Anoop Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *Proceedings of the Eighth ACM International Conference on Supercomputing*, pages 274–288, Manchester, England, July 1994.

[18] 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.

[19] 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.

[20] 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.

[21] 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.