A Comparison of Software and Hardware Synchronization Mechanisms for Distributed Shared Memory Multiprocessors*

John B. Carter, Chen-Chi Kuo, Ravindra Kuramkote

{retrac, chenchi, kuramkot}@cs.utah.edu
WWW: http://www.cs.utah.edu/projects/avalanche

UUCS-96-011

Department of Computer Science
University of Utah, Salt Lake City, UT 84112

September 24, 1996

Abstract

Efficient synchronization is an essential component of parallel computing. The designers of traditional multiprocessors have included hardware support only for simple operations such as compare-and-swap and load-linked/store-conditional, while high level synchronization primitives such as locks, barriers, and condition variables have been implemented in software [9, 14, 15]. With the advent of directory-based distributed shared memory (DSM) multiprocessors with significant flexibility in their cache controllers [7, 12, 17], it is worthwhile considering whether this flexibility should be used to support higher level synchronization primitives in hardware. In particular, as part of maintaining data consistency, these architectures maintain lists of processors with a copy of a given cache line, which is most of the hardware needed to implement distributed locks.

We studied two software and four hardware implementations of locks and found that hardware implementation can reduce lock acquire and release times by 25-94% compared to well tuned software locks. In terms of macrobenchmark performance, hardware locks reduce application running times by up to 75% on a synthetic benchmark with heavy lock contention and by 3%-6% on a suite of SPLASH-2 benchmarks. In addition, emerging cache coherence protocols promise to increase the time spent synchronizing relative to the time spent accessing shared data, and our study shows that hardware locks can reduce SPLASH-2 execution times by up to 10-13% if the time spent accessing shared data is small.

Although the overall performance impact of hardware lock mechanisms varies tremendously depending on the application, the added hardware complexity on a flexible architecture like FLASH [12] or Avalanche [7] is negligible, and thus hardware support for high level synchronization operations should be provided.

*This work was supported by the Space and Naval Warfare Systems Command (SPAWAR) and Advanced Research Projects Agency (ARPA), Communication and Memory Architectures for Scalable Parallel Computing, ARPA order #B990 under SPAWAR contract #N00039-95-C-0018

Contents

1 Introduction **3**

2 System Organization **4**
2.1 Basic Widget Architecture 4
2.2 Support for Multiple Protocols 6

3 Hardware and Software Lock Mechanisms **7**
3.1 Software Locks 7
3.1.1 Ticket Locks 7
3.1.2 MCS Locks (software distributed locks) 8
3.2 Hardware Locks 9
3.2.1 Simple Centralized Hardware Lock Mechanism 10
3.2.2 Ordered Centralized Hardware Lock Mechanism 10
3.2.3 Distributed Hardware Lock Mechanism 10
3.2.4 Adaptive Hardware Locks 11

4 Experimental Methodology **13**
4.1 Simulation Environment 13
4.2 Applications 14

5 Results **16**
5.1 Software vs Hardware Locks 16
5.2 Central vs Distributed Hardware Locks 19

6 Conclusions **21**

1 Introduction

Traditionally, high level synchronization operations have implemented in software using low level atomic hardware primitives such as compare-and-swap and load-linked/store-conditional. The design decisions that led to this split in functionality was driven by the dominant bus-based architectures of previous generation shared memory multiprocessors. On these machines, broadcast invalidations or updates are cheap and the existing memory system is only designed to handle basic loads and stores – specifically, no provision is made in the memory system to maintain lists of which nodes have or want a copy of a cache line.

However, over the past five years, directory-based multiprocessors have emerged as the dominant scalable shared memory architecture [1, 7, 12, 13, 17]. On these machines, communication is expensive and the distributed directory controllers maintain a list of nodes that have copies of each cache line so that they can be invalidated or updated. Furthermore, the recent trend has been to introduce greater intelligence and flexibility to the node memory controllers [7, 12, 17]. Therefore, we believe that it is worth considering supporting for higher level synchronization primitives (e.g., locks, barriers, condition variables, etc.) in hardware. Specifically, we explore the implementation complexity and potential performance impact of adding hardware support for locks.

Many directory protocols have been proposed, but all of them have in common the ability to keep track of a list of nodes that are caching a particular cache line. This ability is almost exactly what is required to implement locks in hardware - a list of nodes waiting for the lock. We show in Section 3 that a variety of hardware lock primitives can be built using existing copyset management hardware. The added flexibility in emerging architectures makes it feasible to implement locks as a special consistency protocol with no additional hardware support whatsoever.

Given the low hardware cost of implementing locks in directory-based multiprocessors, we studied the performance of two software and four hardware implementations of locks to determine whether the provision of hardware locks would significantly impact overall program execution times. We found that hardware support can reduce lock acquire and release times by 25-94% compared to well tuned software locks. In terms of application performance, hardware locks outperform software locks by up to 75% on a synthetic benchmark with heavy lock contention and by 3%-6% on a suite of SPLASH-2 benchmark programs. Given that the added hardware design cost of supporting hardware locks is minimal, even moderate performance improvements are worth pursuing. Furthermore, the synthetic experiments and microbenchmarks indicate that much larger performance improvements are possible for programs with high synchronization to data management ratios.

The rest of the paper is organized as follows. In Section 2, we briefly describe the architecture in our Widget project [7] and the implementation of various fundamental synchronization operations. In Section 3, we present two software lock mechanisms (ticket locks and MCS locks [15]) that can be built on conventional shared memory systems and four hardware lock mechanisms that can be built as extensions to conventional directory controller protocols. In Section 4, we discuss the simulation methodology that we employed to evaluate the various synchronization mechanisms and describe the applications used to evaluate the mechanisms, with special emphasis given to their synchronization patterns. In Section 5, we present the results of our simulation studies and evaluate the tradeoffs between software and hardware synchronization mechanisms, and the advantages of distributed hardware synchronization mechanisms compared to centralized implementations. Finally, concluding remarks and suggestions for future works are provided in Section 6.

2 System Organization

The goal of the Widget project is to develop a communication and memory architecture that supports significantly higher effective scalability than existing multiprocessors by attacking all sources of end-to-end communication latency for both shared memory and message passing architectures. Our approach for achieving this goal is to design a flexible cache and communication controller that tightly integrates the multiprocessor's communication and memory systems and incorporates features designed specifically to attack the problem of excessive latency in current multiprocessor architectures. The focus of the work described in this paper is leveraging the hardware already required for data consistency management to reduce the often high synchronization stall times experienced by shared memory programs. This section provides a brief overview of the features of the Widget most pertinent to the work described herein - further details can be found on the Widget home page (`http://www.somewhere.edu/projects/...`).

2.1 Basic Widget Architecture

A block diagram of an node Widget controller is given in Figure 1. An Widget controller sits directly on HP's Runway memory bus. The Runway bus is a 64-bit multiplexed address/data bus that supports snoopy based multiprocessor coherency protocol. It is a split transaction bus and offers sustainable bandwidth of 768 Mbytes/s at 120 MHz. Each Widget controller contains a Runway Interface and Munger (RIM), a Network Interface (NI), a Shared Buffer (SB), a Cache Controller (CC), a Directory Controller (DC), a Protocol Processing Engine (PPE), and a Bookkeeping Cache (BC).

The RIM is responsible for multiplexing Runway transaction requests from the other components, arbitrating for control of the Runway bus, and transferring data between the Runway bus and the SB. The RIM also implements the logical operations necessary to keep a cache line coherent when partial cache line updates or non-aligned data is received and to send only the dirty words when part of a cache line is dirtied by the local processor. The NI interfaces with the Myrinet interconnect [4] and is responsible for sending and receiving packets. The SB is used to stage the data in transition between local memory and network and also to cache remote shared memory lines and messages until they are invalidated by remote nodes or consumed by the local processor. The BC is used to cache the meta-data needed by the various components. The CC maintains the state and consistency of shared data in the SB and remote data cached in the local memory - Widget will support a variant of the Simple COMA architecture [18] that supports a graceful transition to CC-NUMA when the DRAM cache is experiencing poor page-grained utilization. To improve performance of the node controller, we aggressively split control and data processing, and allow multiple functional units to perform in parallel. Each of the components except the NI snoop the Runway bus so that control processing can be done in parallel with data transfers between the RIM and SB. Similarly, the NI splits the control and data portions of an incoming message so that the receiving component can process the message in parallel with the NI staging the data to the SB. This latter design is similar to the way in which the MAGIC chip splits data and control in the FLASH multi-processor [12].

The key component for the introduction of hardware synchronization mechanisms is the directory controller (DC). On conventional directory-based shared memory multiprocessors, the DC is responsible for maintains the state of the distributed shared memory. Each block of global physical memory has a "home" node and the DC at each home node keeps track of the state of each local cache line, tracking such information as the protocol being used to manage that block of data and the set of nodes that have a copy of the data. As described in the following section, it is trivial to

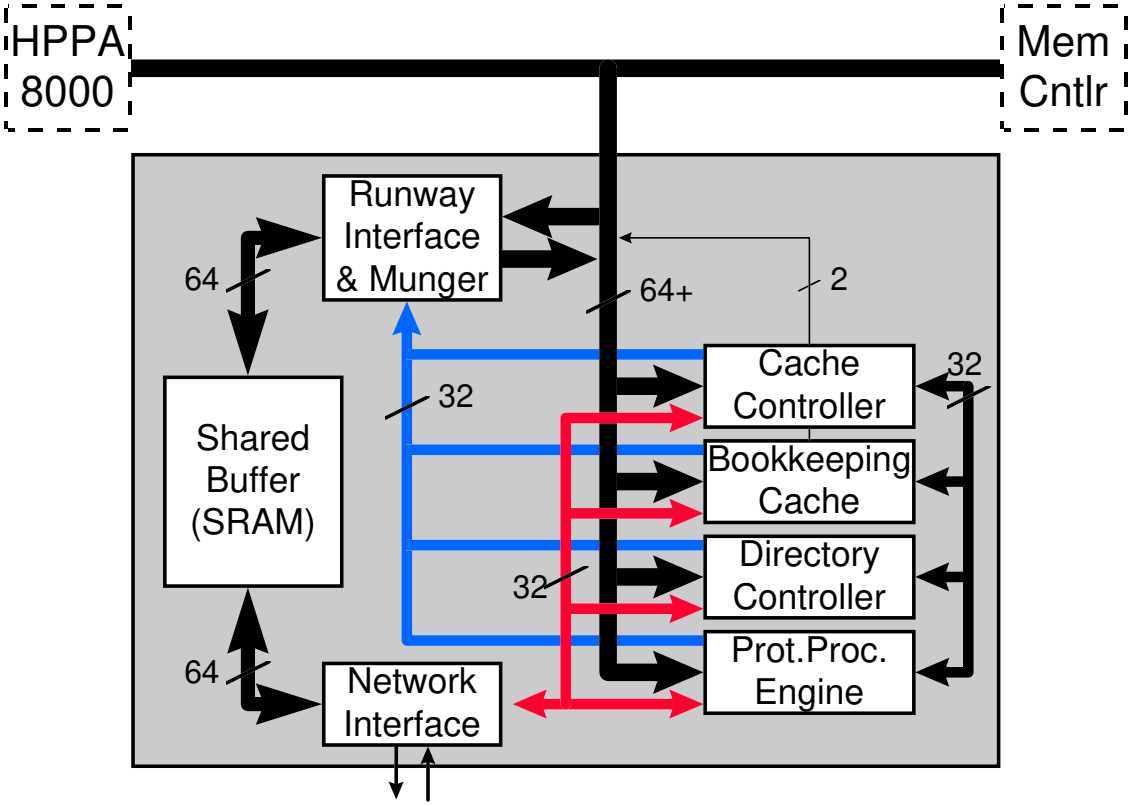

Figure 1: Widget Controller

extend the operation of the directory controller to add support for hardware locks - which we are planning to incorporate in the Widget design.

2.2 Support for Multiple Protocols

Most of the current generation of massively parallel systems support shared memory in hardware (e.g., machines by Convex, Cray, and IBM). However, they all support only a single, hard-wired write-invalidate consistency protocol[1] and do not provide any reasonable hooks with which the compiler or run-time system can guide the hardware's behavior. Using traces of shared memory parallel programs, researchers have found there are a small number of characteristic ways in which shared memory is accessed [3,10,19]. These characteristic "patterns" are sufficiently different from one another that any protocol designed to optimize one will not perform particularly well for the others. In particular, the exclusive use of write-invalidate protocols can lead to a large number of avoidable cache misses when data that is being actively shared is invalidated and subsequently reloaded. The inflexibility of existing machines' cache implementations limits the range of programs that can achieve scalable performance regardless of the speed of the individual processing elements and provides no mechanism for tuning by the compiler or run-time system.

These observations have led a number of researchers to propose building cache controllers that can execute a variety of caching protocols [6,21], support multiple communication models [8,11], or accept guidance from software [12,17]. We are investigating cache designs that will implement a variety of caching protocols, support both shared memory and message passing efficiently, accept guidance from software to tune its behavior, and directly support efficient high-level synchronization primitives. Our goal is to significantly reduce the number of messages required to maintain coherence, the number of cache misses taken by applications due to memory conflicts, and the overhead of interprocess synchronization. We propose to do this by allowing shared data and synchronization protocols to be maintained using the protocol best-suited to the way the programming is accessing the data. For example, data that is being accessed primarily by a single processor would likely be handled by a conventional write-invalidate protocol [2], while data being heavily shared by multiple processes, such as global counters or edge elements in finite differencing codes, would likely be handled using a delayed write-update protocol [3]. Similarly, locks will be handled using appropriate locking protocol in hardware, while more complex synchronization operations reduction operators for vector sums will be handled using specialized protocols in software. By handling data and synchronization with a flexible protocol that can be customized for its expected use, the number of synchronization messages, cache misses and messages required to maintain consistency drop dramatically, as illustrated in the following sections.

In Widget, we support multiple hardware consistency protocols in hardware by introducing "protocol bits" on a per-page basis that tell the Widget node controller how to manage cache lines on that particular page. The protocol used to manage a particular page will be specified by the compiler or operating system, and the protocol bits in the page table are copied to the meta-data of the DC and the CC when the page is created or accessed. The DC and the CC maintain 4 protocol bits per cache line, thus allowing 16 different protocols to be supported directly by the hardware. The DC and the CC have programmable state engines giving flexibility to control the protocol and the performance of hard-wired protocol. Support for high-level synchronization in hardware can be easily implemented by the Widget controller using reads and writes to pages marked as managed via the "lock" protocol and mapped into IO space. To acquire a lock, the application would simply read the lock address in IO space. As part of its normal operation, the CC interprets

[1] Except in the case of the Cray, which does not cache shared data.

the read, determines that it is a read to shared data, and sends a message to the DC at the home node, which returns the lock status (free, busy, ...). The CC state engine takes the appropriate action depending on the protocol and state. To release a lock, the processor simply stores any value to the lock address. Since the CC and DC are already designed to have simple programmable state engines, supporting multiple synchronization protocols in hardware does not add significant design complexity. Architectures such as FLASH [12] and Tempest/Typhoon [17] should be able to support hardware locks with similar ease.
Currently, we are simulating three different shared data consistency protocols - migratory, delayed write update, and write invalidate. The various lock protocols being considered is explained in the following section. We are also considering whether other high level synchronization operations (e.g., barriers) are worthy of hardware support.

3 Hardware and Software Lock Mechanisms

The designers of traditional multiprocessors have included hardware support only for simple operations such as test-and-set, fetch-and-op, compare-and-swap, and load-linked/store-conditional. Higher level synchronization primitives such as locks, barriers, and condition variables have been implemented in software [9, 14, 15]. However, these designs were all originally based on simple bus-based shared memory architectures. Recent scalable shared memory designs have employed distributed directories and increasingly sophisticated and flexible node controllers [7, 12, 17]. In particular, as part of maintaining data consistency, these architectures are already required to maintain lists of processors with a copy of a given cache line, which is most of the hardware needed to implement distributed locks. Given the negligible implementation overhead of supporting locks in these architectures, we believe that it is worthwhile to reconsider whether higher level synchronization primitives should be supported in hardware.
In this section we describe two of the best software lock protocols described in the literature, ticket locks and MCS locks [16] and four hardware lock implementations built on Widget's flexible directory controller.

3.1 Software Locks

Many software synchronization algorithms have been proposed for shared memory multiprocessors [14, 15]. Existing software synchronization algorithms are built on simple hardware atomic operations provided by the hardware designers. A study by Michael and Scott [16] showed that it is possible to achieve good overall synchronization performance on a multiprocessor with a sequentially consistent write-invalidate consistency protocol if great care is taken in the way in which the lock data is laid out in memory and access to this data is carefully managed to avoid undue contention. In particular, ticket locks and queue-based MCS locks can be implemented using only atomic fetch-and-increment, fetch-and-store, and compare-and-swap operations. Although the two algorithms differ slightly in their performance under high and low loads, both avoid much of the overhead found in conventional spin lock implementations caused by excessive invalidation and reload traffic.

3.1.1 Ticket Locks

Michael and Scott [16] described a ticket-based locking system in which each process desiring access to a critical section selects a unique "ticket" value using fetch-and-increment (see Algorithm 1).

After receiving a unique ticket value, each process waiting on the lock spins on whose_turn until their number is set. In our implementation, each lock record has been allocated on a separate cache line. When only one process is contending for the lock, the process performing the fetch-and-increment will obtain an exclusive copy in its cache and the read on whose_turn will be executed locally. However, when multiple processes are contending for a lock, the release will invalidate the shared cache lines in all competing processes' nodes, which introduces a significant amount of communication for both the invalidates and the subsequent reloads. Ticket locks have lower overhead under low contention conditions compared to array-based queuing locks and MCS locks. However, under heavy contention conditions, the number of invalidations and reloads will seriously degrade interconnect performance.
In addition, it is difficult to dynamically choose a generally useful value for proportional_backoff, which is used to reduce the amount of unnecessary traffic and improve the performance of ticket locks under heavy contention conditions. We used 700 cycles and 3000 cycles for proportional_backoff under low and heavy contention conditions respectively. Also, the use of a random backoff means that the ticket lock protocol cannot guarantee FIFO lock acquisition.

```
typedef struct TIXLOCK {
   unsigned long next_tix;
   unsigned long whose_turn; }
tix_lock_type;

void acquire(long lock_addr) {
   unsigned long my_tix;
   tix_lock_type* L = (tix_lock_type*)lock_addr;

   my_tix = fetch_and_increment((long)&(L->next_tix));
   while (L->whose_turn != my_tix)
      pause(proportional_backoff * (my_tix- L->whose_turn));
}

void release(long lock_addr) {
   tix_lock_type* L = (tix_lock_type*)lock_addr;

   L->whose_turn = L->whose_turn+1;
}
```

Algorithm 1: Ticket Lock Algorithm

3.1.2 MCS Locks (software distributed locks)

Mellor-Crummey and Scott introduced a software lock protocol that builds a distributed queue of waiting processes rather than contending for a single counter or flag (see Algorithm 2). The introduction of a distributed queue removes the serious problem seen in ticket locks and spinlocks during periods of heavy contention - releases cause global invalidates that are followed by a flurry of (mostly useless) reloads. Since each process spins on its own personal flag, each release-acquire pair only results in a single invalidation and reload (plus the one required to put a process on to the queue in the first place). During periods of low contention, the extra work required to implement the distributed queue is unnecessary and can significantly increase the synchronization overhead compared to ticket locks. However, MCS locks had the best overall performance in a previous

study of synchronization mechanisms [15], which led Lim and Agarwal to adopt it as their high contention mechanism in their *reactive* synchronization implementation [14]. MCS locks perform best on machines that support both the compare-and-swap and fetch-and-store operations, but it can be implemented using only fetch-and-store, albeit with higher overhead and without FIFO order guaranteed. Our implementation assumes that the underlying hardware provides both primitives. The important thing to note is that each competing process only spins on its local cache line (qnode) until its predecessor reassigns qnode→locked to false.

```
typedef struct qnode {
   struct qnode *next;
   boolean locked; }
qnode_type;

typedef struct MCSLOCK {
   qnode_type *q_ptr; }
mcs_lock_type;

void acquire(long lock_addr) {
   /* allocated to each lock acquire instance */
   /* and keep permanent until release */
   qnode_type* qnode = allocate_qnode(lock_addr);
   mcs_lock_type* ml = (mcs_lock_type*)lock_addr;
   qnode_type *pred;

   qnode->next = NULL;
   pred = (qnode_type *) fetch_and_store((long)&ml->q_ptr, (long)qnode);

   if(pred!=NULL) {
      qnode->locked = TRUE;
      pred->next = qnode;
      while (qnode->locked);
   }
}

void release(long lock_addr) {
   qnode_type* qnode = get_qnode(lock_addr);
   mcs_lock_type* ml = (mcs_lock_type*)lock_addr;

   if (qnode->next == NULL) {
      if (compare_and_swap((long)&ml->q_ptr, (long)qnode, NULL)) return;
      while (qnode->next == NULL);
   }
   qnode->next->locked = FALSE;
}
```

Algorithm 2: MCS Lock Algorithm

3.2 Hardware Locks

In a directory-based distributed shared memory system, the natural place to implement even the basic atomic operations (fetch-and-op, compare-and-swap, test-and-set, etc) is at the directory controller (or memory controller) on the lock's home node. The directory controller is required to

maintain lists of nodes with copies of individual cache lines, and therefore maintaining a list of nodes waiting for a lock requires negligible added hardware complexity. Thus, in directory-based multiprocessors, implementing locks and other higher level synchronization operations does not require significantly increased hardware or firmware complexity. Both hardware and software locks must communicate with the directory controllers managing the lock's cache line, but since conventional data consistency protocols are not well suited to synchronization access patterns [3], the software lock implementations will require extra communication traffic. These observations led us to investigate four directory-based hardware lock mechanisms.

The two simplest hardware mechanisms communicate only with the directory controller on the lock's home node to acquire and release locks. These *centralized* mechanisms differ only in the way in which the manage the list of waiting processes. The third hardware protocol mimics the *distributed queueing* nature of MCS locks, but without the extra software and coherence overhead of the software implementation. The final *adaptive* protocol attempts to switch between centralized and distributed modes based on dynamic access behavior, in a manner analogous to Lim and Agarwal's software reactive locks [14]. All four mechanisms exploit Widget's ability to support multiple consistency protocols by extending the hardware state engine in each directory controller to exchange the required synchronization messages, described below. In each case, we assume that locks are allocated one-per-line in IO space so that all reads and writes can be snooped off of the Runway by the Widget node controller.

3.2.1 Simple Centralized Hardware Lock Mechanism

The DASH multiprocessor's directory controllers maintain a 64-bit copyset for each cache line, where each bit represents one of the 64 DASH processors [13]. This full map directory design can be adapted to support locking by converting acquire requests for remote locks into lock requests to the relevant directory controller. When a directory controller receives a *lock request* message, it either immediately replies with a *lock grant* message if the lock is free or it marks the node as waiting for the lock by setting the bit in its copyset corresponding to the requesting node. When a node releases a lock, it sends a *lock relinquish* message to the corresponding directory controller, which either marks the lock as free if no other node is waiting for the lock, or selects a random waiting processor and forwards the lock to that processor. This implementation is extremely simple, but starvation is possible and FIFO ordering is not maintained.

3.2.2 Ordered Centralized Hardware Lock Mechanism

Architectures, such as Alewife's [1], that manage their directories using linked lists can easily implement a centralized locking strategy that maintains FIFO ordering. For example, in Alewife each directory entry can directly store up to four sharing processors or nodes awaiting a lock. Once the number of contending nodes exceeds four, a software handler is invoked to flush the record of the sharing processors into a software emulated full map buffer. This architecture can implement the centralized scheme described above with FIFO access guarantees. The only disadvantage of this scheme is the software cost for handling copyset overflows. In the FLASH multiprocessor [12], the linked list could easily be managed by the MAGIC chip and stored in the data cache.

3.2.3 Distributed Hardware Lock Mechanism

The ordered centralized lock mechanism guarantees FIFO access ordering, but like all centralized protocols, it can result in serious network and controller hot spots that degrade other transactions

processed by the same node. In addition, because all lock requests are handled by the lock's home node, two messages are required to forward a lock between nodes during periods of heavy contention (one from the lock holder to the home node and one from the home node to the requester). To address these problems, we developed a simple distributed lock mechanism based on an MCS-like distributed queueing model.

In this protocol, each directory controller records only the next requester in the distributed queue - the lock's home node maintains a pointer to the end of the distributed queue. When a process attempts to acquire a lock, it sends a *lock request* message to the lock's home directory controller, which either grants the lock immediately if it is free or forwards the request to the node at the end of the distributed queue while updating its tail pointer (see Figure 2). If when a process releases a lock it has already received a *lock request* message, it immediately forwards the lock to that node without first informing the lock's home node. During periods of heavy load, this approach reduces the number of messages required to acquire a lock from four to three. In addition, if a lock has a high degree of reuse by a particular node, that node will be able to reacquire the lock with no communication whatsoever during periods of low contention (see Figure 3). Compared to software queue-based (MCS) locks, this hardware mechanism does not add overhead during periods of low contention, and the number of messages required to forward a lock are reduced from seven (four to invalidate it and three to reload it) to two.

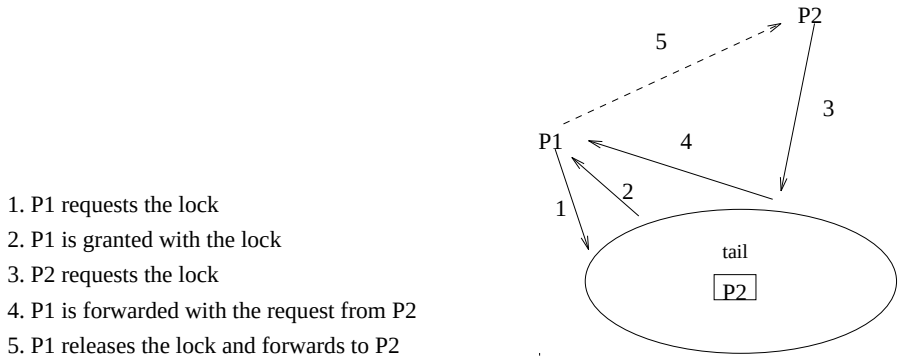

Figure 2: Example of forwarding requests in the distributed hardware lock mechanism

3.2.4 Adaptive Hardware Locks

Because the centralized and distributed lock mechanisms perform quite differently during periods of heavy and light contention, we developed an adaptive hardware lock scheme. The basic idea of the adaptive protocol is to adopt a centralized scheme during periods of low contention and switch to a distributed protocol during periods of heavy contention. For each directory entry in the directory controller, a list of the first four nodes waiting on the lock are stored in the lock's directory entry, as in Alewife [1]. As long as no more than four nodes are waiting on the lock at a time, it remains in low contention (centralized) mode - releases go back to the home node and the lock cannot be cached. However, instead of handling directory entry overflow by trapping to software, we instead switch to the distributed mechanism, as illustrated in Figure 4. We remain in distributed mode until the burst of heavy contention passes and the lock becomes free. When a processor releases the lock and finds that no other node is waiting for it, it sends a *release hint* message to the home node to propose a return to centralized mode, as illustrated in Figure 5. If the home node has not

1. P2 releases the lock,
   but still holds the lock
2. P3 requests the lock
3. P2 reacquires the lock
4. P2 is forwarded with the request from P3
5. P2 releases the lock
6. P2 forwards the lock to P3

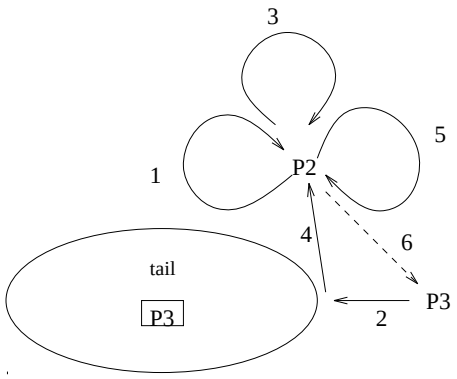

Figure 3: Example of lock reuse in the distributed hardware lock mechanism

1. P1 requests the lock
2. P1 is granted with the lock
3. P2 requests the lock
3.5. P1 releases the lock
4. P3 requests the lock
5. P4 requests the lock
6. P3 is forwarded with the request from P4
7. P5 requests the lock
8. P4 is forwarded with the request from P5
t. P3 releases the lock and forwards to P4
t+1. P4 releases the lock and forwards to P5

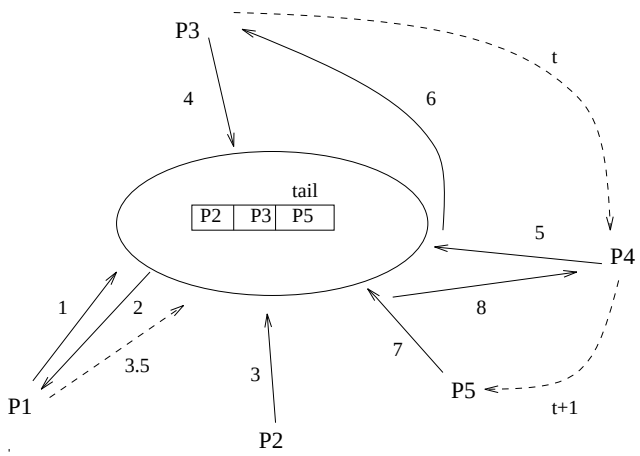

Figure 4: Example from central scheme to distributed scheme

forwarded any recent requests, it will respond with a *release acknowledge* message at which time the protocol will revert to the centralized scheme. Until the acknowledgement arrives, the last node to acquire the lock will continue to respond to lock requests to avoid race conditions.

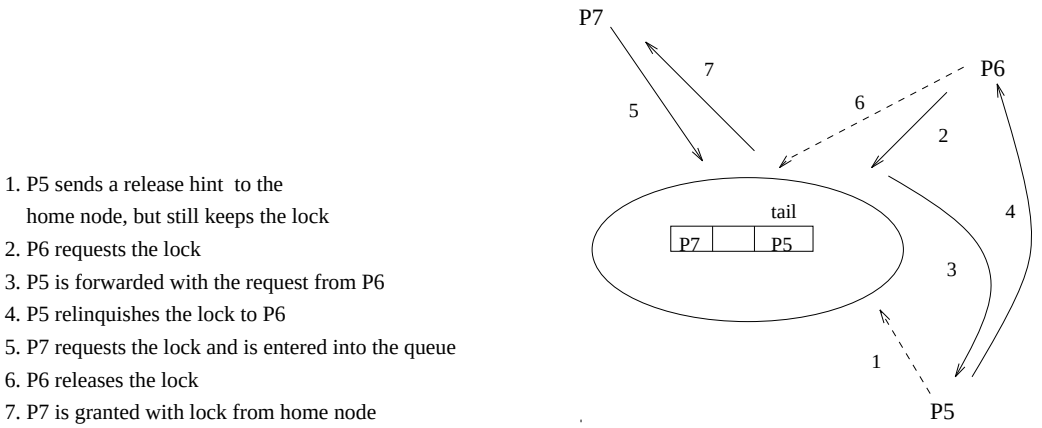

Figure 5: Example from distributed scheme to central scheme

4 Experimental Methodology

4.1 Simulation Environment

To evaluate a wide variety of possible synchronization implementations, we used the PAINT simulation environment, a locally ported HP PA-RISC version of the MINT multiprocessor memory hierarchy simulation environment [20]. PAINT simulates a collection of processors and provides support for spinlocks, semaphores, barriers, shared memory, and most Unix system calls. It generates multiple streams of memory reference events, which we used to drive our detailed simulation model of the Widget multiprocessor memory and communication hierarchy described in Section 2. We replaced the built-in spinlocks with our synchronization protocols as described in Section 3.
Our experience and that of others [5] is that accurate performance measurements of a multiprocessor's backplane is critical for effectively evaluating the true performance of DSM architectures. Therefore, we have developed a very detailed, flit-by-flit model of the Myrinet network fabric [4] to accurately model network delays and contention. The Myrinet fabric is mesh-connected, with one crossbar at the core of each switching node. Our network model accounts for all sources of delay and contention within the network for each flit of data including per-switching-node fall through times, link propagation delays, contention for the crossbars in each switching node, and contention for the input and output FIFOs in each compute and switching node. Because the performance of the Myrinet fabric will soon be increased to be competitive with alternative multiprocessor interconnects and to avoid biasing our results in favor of hardware synchronization, we simulated a network with one-tenth of Myrinet's high fall through latency (4 cycles vs 40 cycles). This decision benefits the software protocols, as they generate approximately two to six times more synchronization-related network traffic than the hardware protocols.
We made the following simplifying assumptions in the model we simulated:

- Each node contains a 256-kilobyte first level cache. In order to reduce the effect of varying

conflict misses between different runs, the cache was 4-way set-associative.

- Only four components of the Widget controller shown in Figure 1 were modeled - the RIM, the CC, the DC, and the NI. The RIM only did bus arbitration, and the DC was assumed to have enough memory to store all needed metadata.
- Each of the four components handled only one request at a time. The contention that this design entails was modeled using a FIFO input and output buffer in each of the component.
- The bus was a split transaction, 32 bit multiplexed address/data bus.
- Table 2 lists the delay characteristics that we used in our model. We based these times on the existing PA-RISC 7100 implementation and our estimate of the time to perform operations within the Widget controller.

Since our study evaluates various hardware and software synchronization protocols rather than detailed node controller design alternatives, the above simplifications do not affect the results presented in this paper. Our model is sufficient to analyze the effect of cache misses, and contention between shared data, control, and synchronization messages. Data was kept coherent using a release consistent invalidation protocol. Space constraints make it impossible to discuss all of the details of the simulation environment herein - a more detailed description of the Widget architecture can be found elsewhere [7].

Operation	Delay
Local read hit	1 cycle
Local write hit	1 cycle
DRAM read setup time	6 cycles
DRAM write setup time	2 cycles
Time to transfer each subsequent word tofrom DRAM	1 cycle
DRAM refresh (time between DRAM requests)	3 cycles
Enqueue a message in a FIFO between controllers	1 cycles
Dequeue a message from a controller's input FIFO	1 cycle
Update directory entry	4 cycle
Interconnect latency per hop	8 cycles

Table 1: Delay Characteristics

4.2 Applications

We used four programs in our study, global counter, barnes, fmm, and radiosity. Figure 6 contains the parameters and a brief description for each of these test program. The global counter program has been used in couple of previous studies [14, 16]. The program has one global counter protected by a lock. All participating processes compete for the lock, update the counter, wait for a period of time for the shared lock, and compete for the next run. We have controlled the degree of contention by using two parameters: the latency of the critical section and the duration between a lock release and the subsequent lock acquire. In addition, the number of participating processes adds another dimension to the analysis. In the ensuing discussion, counter-n represents the global

counter program with 500 cycles in each critical section and n cycles between critical sections. barnes, fmm, and radiosity are from the SPLASH-2 benchmark suite [22]. Table 2 contains the average duration of critical sections and the average time between acquire attempts for each of these programs. In barnes, a cell lock is allocated to protect each space cell, a global lock protects the global maximum and minimum values, and a global IO lock protects standard output. During each time step, all processes wait on a barrier at the start and then load the bodies into a tree structure. This phase of the program uses the maximum number of locks. The degree of contention increases as the number of processes is increased without changing the problem size. After the local structures are initialized, a significant amount of time is spent in the computation phase. After the computation phase, each process updates a global structure that is protected by the global lock. Contention for the global lock is not as high as that for cell locks, because of irregular load distributions. In barnes the tree is traversed once for every particle, while the processes in fmm perform only one upward pass followed by a downward pass for each cell during each time step. The result is propagated to the bodies in the downward pass. Therefore, the average duration of critical section is longer, but the contention is lower than that in barnes. radiosity uses distributed task queues, which are protected by locks, for parallelism. Each process manages its own local distributed task queue, which is accessed by other processes infrequently. Therefore, the contention in radiosity is the lowest among all the applications we considered.

Application	Parameter	Description
Global counter	50 cycle critical section with 500 to 30000 cycles between critical sections. Total of 600 lock acquire and release by each process.	A global lock is used to protect a global counter
Barnes	1024 nbodies	Simulates the interaction of a system of bodies in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method.
Fmm	1024 nbodies	Simulates a system of bodies in two dimensions over a number of time-steps, using the adaptive Fast Multipole Method.
Radiosity	The small test scene	Computing the equilibrium distribution of light in a scene using interactive hierarchical diffuse radiosity method.

Figure 6: Description of Applications

# of nodes	4	8	16	32
Global Counter 1	2400/50/500	4800/50/500	9600/50/500	N/A
Global Counter 2	2400/50/30k	4800/50/30k	9600/50/30k	N/A
Barnes(1024)	4360/581.43/43.5k	4425/904.81/47.2k	4557/1322.57/52.1k	4762/1894.11/67.9k
Fmm(1024)	4484/952.22/33.7k	4593/1456.21/46.5k	4788/2488.20/74.9k	N/A
Radiosity	282797/135.20/6.2k	270995/139.63/6.5k	276154/136.16/7.3k	289894/135.08/11.6k

Table 2: Number of locks, average critical section in processor cycles, average lock reacquire attempt cycle (k=1000) (based on MCS lock running with release invalidation coherence protocol).

# of nodes	4	8	16	32
Barnes(1024)	4362/294.24/36.3k	4428/291.17/37.2k	4593/284.60/37.1k	4843/276.59/36.6k
Fmm(1024)	4476/361.26/26.8k	4573/356.94/30.6k	4809/346.58/41.4k	N/A
Radiosity	283468/84.00/3.5k	283196/87.34/3.6k	276642/84.87/4.3k	289904/84.82/6.9k

Table 3: Number of locks, average critical section cycle, average lock reacquire attempt cycle (k=1000) (based on MCS lock with 1 cycle data access time for all kind of regular data accesses).

5 Results

In this section we compare the performance of the software and hardware lock implementations on a variety of programs. In addition, we evaluate the various hardware lock mechanisms to determine which, if any, perform particularly well or poorly.

5.1 Software vs Hardware Locks

Table 4 presents the average time to perform an acquire and a release for each of the applications and each of the six lock protocols on sixteen nodes. The four hardware protocols all require significantly less time (25-94%) to acquire and release locks compared to the software protocols. This dramatic difference in raw synchronization latency confirms our suspicion that specialized hardware synchronizatin implementations have great potential in distributed shared memory multiprocessors.

App	counter500	counter30k	barnes1024	fmm1024	radiosity
MCS	100.00%/100.00%	100.00%/100.00%	100.00%/100.00%	100.00%/100.00%	100.00%/100.00%
Tic	147.96%/128.46%	195.22%/95.14%	163.67%/73.25%	155.58%/106.67%	269.10%/63.33%
Ran	15.86%/7.30%	9.15%/6.40%	66.70%/11.93%	66.62%/10.18%	53.16%/43.33%
Ord	29.43%/7.30%	9.47%/6.40%	73.55%/11.93%	66.08%/10.18%	52.49%/43.33%
Dis	22.22%/7.30%	11.16%/6.40%	69.29%/11.93%	67.27%/10.18%	61.46%/43.33%
Ada	22.22%/7.30%	9.47%/6.40%	74.34%/11.93%	63.64%/11.23%	52.49%/43.33%

Table 4: Average Acquire and Release Times (16 nodes)

In terms of impact on overall application performance, Figures 7 and 8 show the simulation elapsed

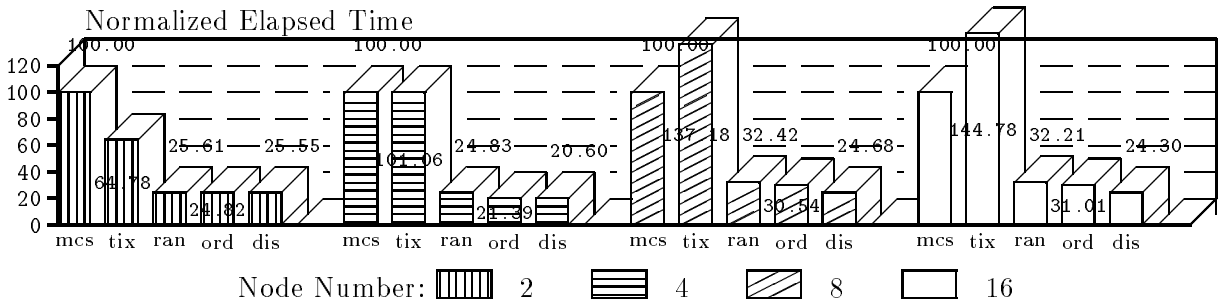

Figure 7: Global Counter Performance Comparison

time for each protocol running the global counter program, normalized to the running time using software MCS locks. We ran two versions of the global counter program: counter500, which represents programs with heavy contention for the same lock, and counter30000, which represents tuned programs with relatively low contention for a given lock. Under heavy lock contention, the hardware locking schemes perform significantly better than the software schemes across the board, reducing executing time by up to 75%. Thus, for programs with high degrees of lock contention or during periods of contention, the provision of hardware locking can have a dramatic impact on performance. Also, under these load conditions, the distributed hardware lock scheme performs up to 25% better than the centralized schemes. Under light load conditions, ticket locks perform significantly better than MCS locks, as expected, although as more nodes are added, their performance deteriorates rapidly. The good performance of the hardware lock schemes, on the other hand, is independent of the number of nodes in the system. The hardware lock schemes outperform the software lock schemes up to 15% in the 16-node case. However, the performance difference between the centralized lock schemes and the distributed lock schemes is not as significant as in the highly contended counter500 program.

In Figure 9, we present the performance of the various locking mechanisms on the 1024-body barnes program. Like in counter30000, the performance of the hardware schemes improves as the number of nodes increases, peaking at 6.24% improvement for the distributed lock protocol for sixteen nodes (over 10% compared to the ticket lock protocol). Because of the low level of lock contention in barnes, the difference between the centralized and and distributed hardware schemes is small.

As illustrated in Figure 10 and Table 2, the amount of contention for locks in fmm is very similar to that seen in barnes. However, the critical sections and the time between lock requests are longer in fmm, which reduces the potential performance impact of the hardware lock implementations. Nevertheless, they manage to outperform the software lock schemes by 5% in the 16-node case[2].

From Figure 11 it is clear that radiosity has much different synchronization characteristics from barnes and fmm. As described in Section 4, radiosity has a significant number of locks and each protects a fairly fine grain data and local access task queues. Processes do not interact, so contention for locks between processes is very infrequent. Therefore, the centralized lock schemes significantly outperform the distributed lock schemes for 4-node and 8-node cases. However, once the number of nodes reaches sixteen, the distributed hardware lock outperforms the centralized lock schemes due to the inherently greater contention caused by the smaller amount of work per processor.

All of the SPLASH-2 programs have been carefully tuned to avoid performing synchronization,

[2]We were unable to run fmm on 32 nodes due to a problem in the simulation environment – something we plan to correct before the final paper deadline.

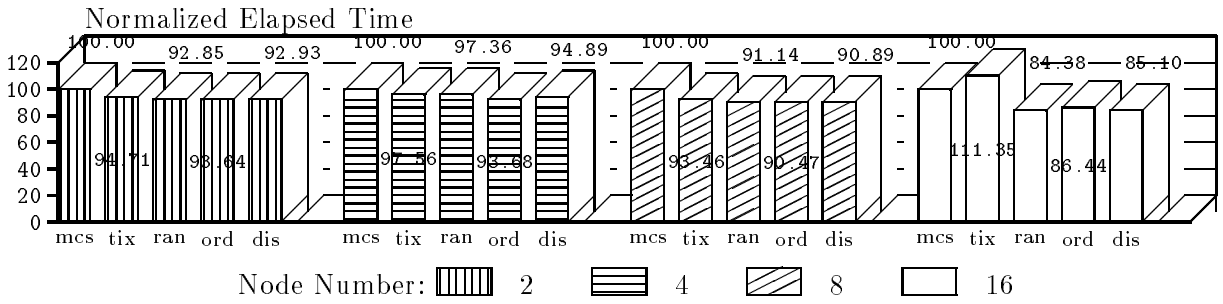

Figure 8: Global Counter Performance Comparison

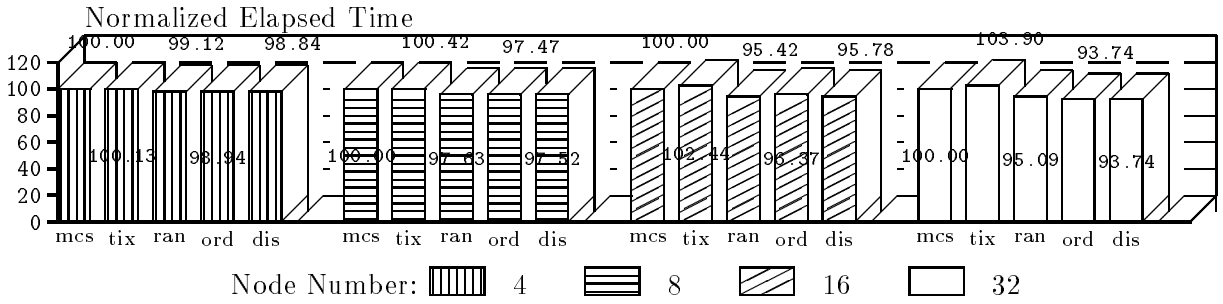

Figure 9: Barnes-Hut with 1024 bodies

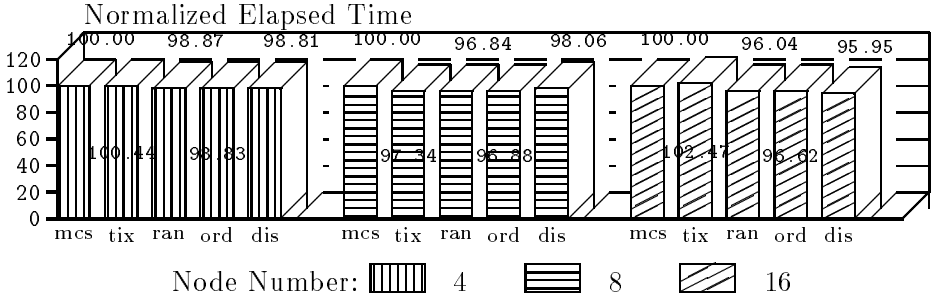

Figure 10: FMM with 1024 bodies

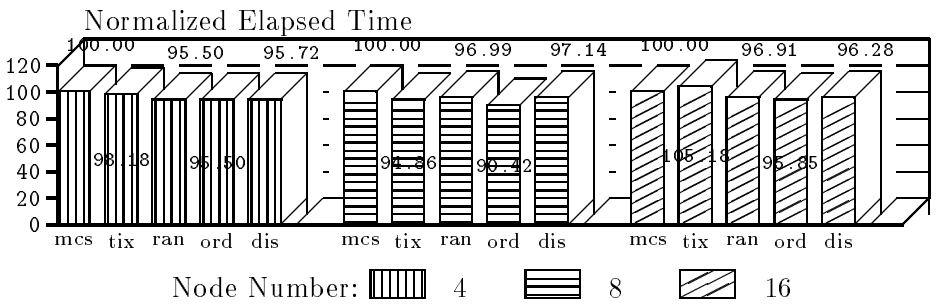

Figure 11: Radiosity

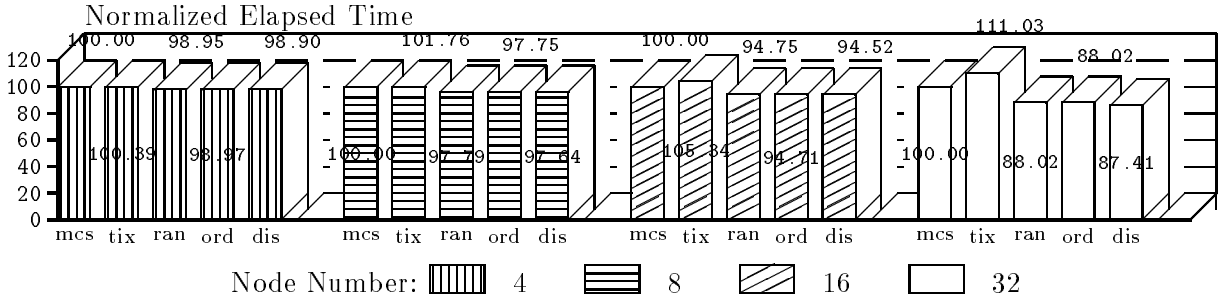

Figure 12: Barnes-Hut with 1024 bodies, fixed data access time

which limits the degree to which any synchronization mechanism can impact overall application performance. Also, all of the data in our applications is managed via a conventional write-invalidate protocol, which can lead to a large amount of excess communication for programs with high degrees of sharing. Therefore, to investigate the impact of hardware synchronization support for programs with higher synchronization-to-computation ratios than is found in the tuned SPLASH-2 programs, we reran the simulations assuming that all shared data accesses took only one cycle (i.e., there were no coherence misses). Clearly this is unrealistic in real programs, but our intent was to isolate the synchronization overhead from the shared data management overhead. Figures 12 through 14 illustrate the performance of the various synchronization primitives when the impact of synchronization is less masked by data management.

5.2 Central vs Distributed Hardware Locks

We observed three major characteristic synchronization access patterns in the applications: low contention with long periods between reacquires by the same process, low contention with repeated reacquires by a single process, and heavy contention by many processes.

In the first scenario, the distributed lock mechanism performs relatively poorly because lock acquire requests must be forwarded from the home node to the last lock holder, who is usually not reacquiring the lock. This requires an extra message and greater controller overhead. In central lock scheme, on the other hand, requests usually can be serviced as soon as they arrive at the home directory, which reduces the average number of messages by one.

In the second scenario, when the contention is low and lock reacquires are frequent such in radiosity,

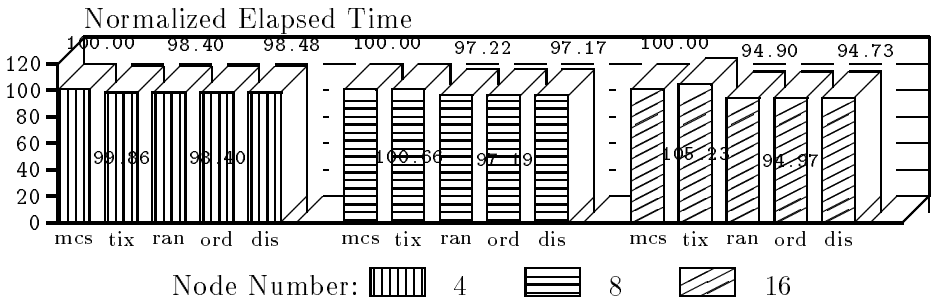

Figure 13: FMM with 1024 bodies, fixed data access time

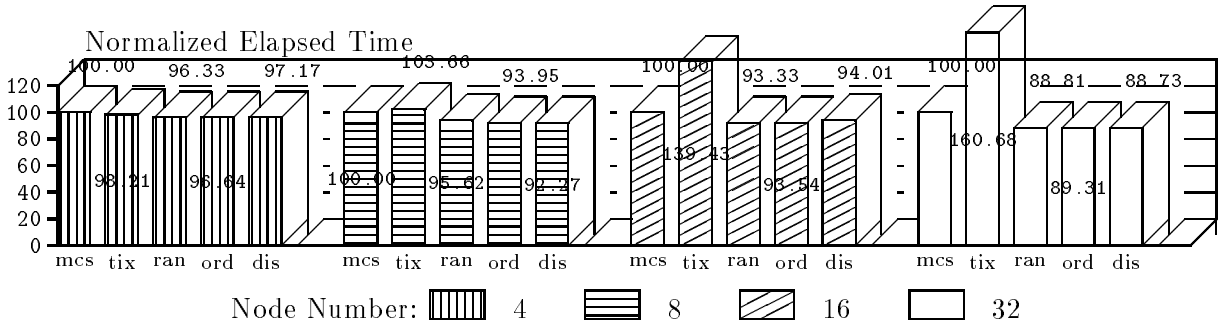

Figure 14: Radiosity, fixed data access time

the relative performance of the centralized and distributed lock mechanisms reversed.
The distributed locking mechanisms ability to cache locks at the site of the last lock request means that all reacquires occur at essentially no cost. In the central lock scheme, each acquire and release must go through the home directory even if the same node is repeatedly reacquiring a lock in a short period of time. This behavior results in significantly higher lock acquire latencies and a much heavier communication load on the network.
In the final scenario, when contention is very high but there is not a large amount of locality between two consecutive acquires, the distributed lock mechanism outperforms the centralized lock mechanism because of the directory controller hot spot and the fact that lock forwarding requires an extra message on the average. In the distributed lock scheme, instead of waiting in the directory, the request is forwarded to the previous process in the distributed linked list, and it can be granted via a single message once the previous process releases the lock.
In Figure 15, the elapsed times for the ordered centralized, distributed, and adaptive hardware locks are shown, relative to the ordered centralized scheme. For most applications and number of nodes, the difference in performance between the various hardware protocols is neglible. Only in the case of the high contention counter500 does the performance vary dramatically – in this case, the distributed and adaptive lock scheme significantly outperform the centralized lock scheme. In the other programs, the relative performance of the centralized and distributed schemes tracked the lock reacquisition rate almost exactly. Since this rate is highly program and problem size dependent, neither scheme clearly dominates the other in terms of performance. Somewhat surprisingly, the adaptive protocol does not perform particularly well (or poorly) for any of the programs. These results indicate that any adaptation should be aimed at detecting periods of frequent lock reacquisition rather than generally high levels of contention. Overall, however, it appears that while the hardware locks consistently outperform the software locks by a significant margin, the difference in performance between the various hardware implementation is generally small, so the choice of hardware protocol should probably be driven by implementation complexity rather than performance.

6 Conclusions

Traditionally, high level synchronization operations have implemented in software using low level atomic primitives such as compare-and-swap and load-linked/store-conditional. This design makes sense on bus-based multiprocessors, since the hardware required to implement locks or barriers would be non-trivial and the payoff small. However, the emergence of directory-based distributed shared memory multiprocessor architecture in the past decade makes hardware synchronization mechanisms more attractive. Many directory protocols have been proposed, all of which can implement at least one of the hardware synchronization protocols described in Section 3. The directory controllers in these architectures already provide the queueing or copyset logic needed to implement hardware locks, and the higher communication costs present in these systems make even small protocol enhancements important.
Given the low hardware cost of implementing locks in directory based multiprocessors, we studied the performance of two software and four hardware implementations of locks to determine whether the provision of hardware locks would significantly impact overall program execution times. We found that hardware implementation can reduce lock acquire and release times by 25-94% compared to well tuned software locks. In terms of macrobenchmark performance, hardware locks reduce application running times by up to 75% on a synthetic benchmark with heavy lock contention and by 3%-6% on a suite of SPLASH-2 benchmarks. As improved cache coherence protocols reduce the

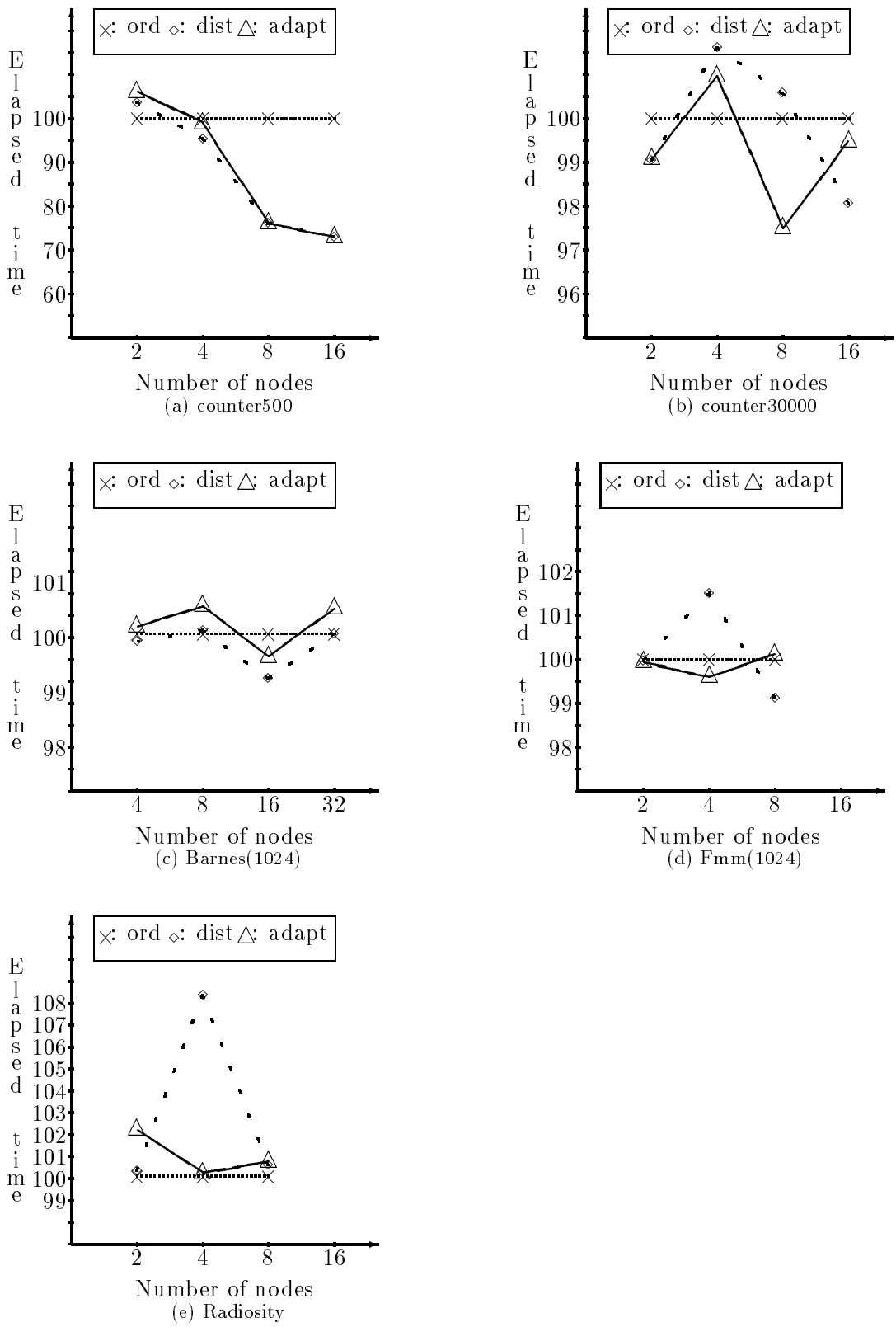

Figure 15: Relative Performance of Hardware Lock Mechanisms

amount of time spent managing shared data relative to the time spent synchronizing, the hardware schemes will have an increasingly significant impact on performance. To examine this trend, we ran several simulations with artificially reduced shared data management overheads. In these simulations, the hardware schemes outperformed the software schemes by up to 21% on the SPLASH-2 benchmark programs. Given Widget's flexible support for multiple cache coherence protocols, the added hardware design cost of supporting hardware locks is minimal, so even moderate performance improvements are worth pursuing. Furthermore, the synthetic experiments indicate that much larger performance improvements are possible for programs with higher synchronization to data management ratios.

We studied four hardware lock mechanisms: two centralized mechanisms (one using an unordered bitmap and another with a local linked list), one that implemented a distributed linked list, and one that switched between the centralized and distributed algorithms depending on dynamic access patterns. The centralized lock schemes perform better when lock contention is low, either due to a small number of competing processes or low rate of interference. As the number of competing processes or rate of interference increases, the distributed lock schemes perform best. However, the overall difference in performance of the four hardware locking schemes is small, so the choice of which hardware mechanism to support should be driven more on ease of implementation than on performance.

References

[1] A. Agarwal and D. Chaiken et al. The MIT Alewife Machine: A large-scale distributed-memory multiprocessor. Technical Report Technical Memp 454, MIT/LCS, 1991.

[2] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[3] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, May 1990.

[4] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su. Myrinet – A gigabit-per-second local-area network. *IEEE MICRO*, 15(February):29–36, February 1995.

[5] D.C. Burger and D.A. Wood. Accuracy vs performance in parallel simulation of interconnection networks. In *Proceedings of the ACM/IEEE International Parallel Processing Symposium (IPPS)*, April 1995.

[6] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[7] J.B. Carter and R. Kuramkote. Avalanche: Cache and DSM protocol design. Technical report, University of Utah, April 1995.

[8] D. Chaiken and A. Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324, April 1994.

[9] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU multicomputer - designing a MIMD shared-memory parallel machine. *IEEE Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[10] A. Gupta and W.-D. Weber. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[11] M. Heinrich and J. Kuskin et al. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[12] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, May 1994.

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

[14] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 25–35, October 1994.

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, February 1991.

[16] Maged M Michael and Michael L. Scott. Scalability of atomic primitives on distributed shared memory multiprocessors. Technical Report 528, University of Rochester Computer Science Department, July 1994.

[17] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[18] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple coma. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 276–285, January 1995.

[19] J.E. Veenstra and R.J. Fowler. A performance evaluation of optimal hybrid cache coherency protocols. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 149–160, September 1992.

[20] J.E. Veenstra and R.J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *MASCOTS 1994*, January 1994.

[21] A. Wilson and R. LaRowe. Hiding shared memory reference latency on the GalacticaNet distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 15(4):351–367, August 1992.

[22] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.