MP-LOCKs: Replacing Hardware Synchronization Primitives with Message Passing *

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

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

UUCS-98-021

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

November 5, 1998

Shared memory programs guarantee the correctness of concurrent accesses to shared data using interprocessor synchronization operations. The most common synchronization operators are locks, which are traditionally implemented in user-level libraries via a mix of shared memory accesses and hardware synchronization primitives like *test-and-set*. In this paper, we argue that synchronization operations implemented using fast message passing and kernel-embedded lock managers are an attractive alternative to dedicated synchronization hardware. We propose three message passing lock (*MP-LOCK*) algorithms (centralized, distributed, and reactive) and provide guidelines for implementing them efficiently. MP-LOCKs reduce the design complexity and runtime occupancy of DSM controllers and can exploit software's inherent flexibility to adapt to differing applications lock access patterns. We compared the performance of MP-LOCKs with two common shared memory lock algorithms: *test-and-test-and-set* and *MCS* locks and found that MP-LOCKs scale better. For machines with 16 to 32 nodes, applications using MP-LOCKs ran up to 186% faster than the same applications with shared memory locks. For small systems (up to 8 nodes), MP-LOCK performance lags shared memory lock performance due to the higher software overhead. However, three of the MP-LOCK applications slow down by no more than 18%, while the other two slowed by no more than 180%. Given these results, we conclude that locks based on message passing should be considered as a replacement for hardware locks in future scalable multiprocessors that support efficient message passing mechanisms. In addition, it is possible to implement efficient software synchronization primitives in clusters of workstations by using the guidelines we proposed.

Keywords: Distributed shared memory, multiprocessor computer architecture, synchronization, lock, unlock.

*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

1 Introduction

Shared memory has become an increasingly popular paradigm for writing parallel programs. One of the purported advantages of shared memory compared to message passing is that it is easier to program. Programmers are not forced to track the location of every piece of data that might be needed. However, to guarantee semantic correctness, shared memory programs must control concurrent accesses to shared data via synchronization operations, the most common of which are *lock* and *unlock*. Without an *efficient* implementation of synchronization, fine-grained parallelism is impossible. An inefficient implementation of synchronization impacts the performance of shared memory programs both directly, via the time required to perform the synchronization operations, and indirectly, by increasing the amount of time processes are blocked waiting for other processes to relinquish locks. As a general rule, multiprocessor architects tend to implement primitive operations using custom hardware. For example, lock and unlock operations traditionally have been implemented using a combination of hardware-implemented shared memory and atomic synchronization primitives (e.g., *test-and-set* (T&S), *compare-and-swap*, and *load-linked/store-conditional* [4, 6, 7, 9]). Recently, however, the designers of the Cray T3E[23] broke this rule and abandoned the dedicated high performance hardware barrier network that was supported in the T3D. Their stated reason for this move was, "We have yet to encounter an application in which barrier time is a large fraction of total run-time, and the dedicated barrier network is expensive...". We believe a similar argument applies to locks and that emerging high performance message passing mechanisms make locks based on message passing a viable alternative to hardware locks in future scalable multiprocessors.

Test-and-Set (T&S) locks [7] spin on shared memory locations using hardware T&S instructions until the previous lock holder releases the lock by writing a "0" in to the lock variable. The major problem with T&S locks is that every T&S instruction performed while spinning involves *global* communication. *Test-Test-and-Set* (T&T&S) locks [21] add an extra shared-memory "load" before the T&S primitive to eliminate these unnecessary migrations by delaying the execution of these global T&S instructions until the previous lock holder frees the lock. However, if multiple nodes are waiting for a lock to be released, all of their local copies of the cache line holding the lock are invalidated as part of the unlock operation. This leads to a flurry of global coherence traffic as each node reloads the lock and performs its lock test, which itself invalidates all remote copies of the lock. Inserting a backoff delay between subsequent T&S instructions reduces the impact of this lock contention, thereby alleviating occupancy problems in the controllers and reducing average lock acquisition latency [21].

Mellor-Crummey and Scott (MCS) [16] locks avoid global spinning entirely by maintaining a distributed queue of processes waiting for a lock. The proposed implementation uses a combination of shared memory and *fetch-and-store* and/or *compare-and-swap* instructions to implement the distributed queue in such a way that waiting processes spin only on a local shared memory location. When a lock is released, only the next process in the wait queue, if any, will perform a coherence operation. In addition to improving performance, this design also guarantees that locks are granted in FIFO order. The downside of MCS locks is that they involve more global operations than T&T&S locks when a lock is free.

Because the performance of T&T&S and MCS locks are heavily dependent on the lock access patterns of cooperative parallel processes, Lim and Agarwal proposed an adaptive scheme called *Reactive* locks [14] that adopts either T&T&S or MCS lock semantics depending on the degree of lock contention.

Finally, the Queue-On-Lock-Bit (QOLB) mechanism [5] associates a special lock bit with cache lines of data. QOLB allows applications to queue waiting for the data to be unlocked, similar to full and empty bits, and was included as part of SCI specification [8]. QOLB's primary advantage over the above lock mechanisms is that it naturally collocates data with the locks that protect it, thereby potentially reducing the amount of communication necessary to perform a critical section. While a complete hardware implementation of QOLB may be the most efficient lock mechanism proposed, it requires significant changes to processor, cache controller, and DSM protocol engine designs. In particular, QOLB requires specialized non-blocking `EnQOLB` and `DeQOLB` processor instructions and changes to processor cache and DSM controller designs so that spinning on the "shadow" copies of data does not trigger coherency transactions. Because they are unsuitable for implementation on multiprocessors based on current commodity microprocessors, we do not consider QOLB locks further.

Unfortunately, conventional shared memory lock mechanisms (T&T&S, MCS, and Reactive locks) were designed based on the performance characteristics of bus-based architectures. On these machines, broadcast invalidations or updates are cheap. However, as remote access becomes increasingly expensive in scalable architectures and as the gap between the speeds of processor and I/O widens, other approaches must be considered.

T&T&S, MCS, and Reactive locks need a combination of hardware atomic primitives and shared memory. Although support for basic shared memory operations is a fundamental part of building a DSM multiprocessor, supporting hardware synchronization operations requires extra hardware resources and design effort (e.g., special registers and additional state machines in the DSM controller). However, many contemporary multiprocessor architectures support efficient message

passing in addition to shared memory [11, 12, 20], and an increasing number of high performance of network interfaces and protocols have been proposed [1, 3, 19, 26, 28]. We propose that this low latency message passing support be exploited to implement synchronization, rather than using hardware synchronization primitives. We call this method of implementing lock primitives via message passing *MP-LOCK*.

Instead of using primitive synchronization operations and shared memory to acquire and release locks, MP-LOCKs send messages to lock managers, which mediate lock requests. In our implementation of MP-LOCKs, the lock managers are embedded inside the operating system kernel to guarantee fast responses to lock requests. In addition to eliminating the design overhead of supporting scalable synchronization primitives in hardware, software's inherent flexibility can be exploited to provide different implementations of locks (e.g., T&T&S style, QOLB style, etc.). MP-LOCKs also can minimize the number of network transfers required to transfer lock ownership, because they do not suffer from unnecessary invalidations and reloads due to the use of general purpose shared memory protocols. Furthermore, MP-LOCKs offload work from the DSM controller to the network controller, thereby reducing DSM controller occupancy. Finally, MP-LOCK's software implementation makes prefetching data (as in QOLB) feasible without the need for non-standard hardware, which should shrink critical sections [10].

To evaluate the tradeoffs of implementing locks in software, we compared the performance of message passing locks with T&T&S and MCS locks on five application programs with fairly heavy synchronization requirements. Although most previous locking studies have concentrated on microbenchmarks, we focused on complete applications to determine the overall impact of using the various locking mechanisms. We found that message passing locks scale better - for machines with 16 to 32 nodes, applications using MP-LOCKs ran up to 186% faster than the same applications with shared memory locks. For small systems (up to 8 nodes), MP-LOCK performance lags shared memory lock performance due to the higher software overhead. However, three of the MP-LOCK applications slow down by no more than 18%, while the other two slowed by no more than 180%.

The remainder of this paper is organized as follows. In Section 2 we describe T&T&S locks, MCS locks, and MP-LOCKs in detail. Section 3 presents some implementation issues involved in making MP-LOCKs efficient. We describe our simulation environment, test applications, and experimental setup in Section 4, and present the results of our experiments in Section 5. In Section 6, we discuss related work. Finally, we draw conclusions and discuss future work in Section 7.

2 Background

In this section, we discuss three implementation of shared memory locks (*T&T&S*, *MCS* and *Reactive*) and three implementations of MP-LOCK (*centralized*, *distributed* and *reactive*) protocols. The shared memory locks in our study are based a write invalidate coherence protocol.

We classify lock acquire into three categories depending on the state it is in: *remote idle acquire*, *local idle acquire*, and *remote busy acquire*. A lock is in *remote idle acquire* state when node Y attempts to acquire the lock after some other node X has released it. The acquire in this case will succeed after the remote operations to detect and update the lock status are completed. The *local idle acquire* state is the same as the *remote idle acquire* state, except X and Y are the same nodes. Both of the idle cases occur frequently when there is less contention. A lock is in *remote busy acquire* state when acquire from node Y is issued before the current lock holder, a different node X, releases the lock. This occurs frequently when there is high contention for locks. We found that performance of an implementation primarily depends on two factors: the number of messages (what we call *hops*) that must be sent to acquire or release a lock when the lock is in each of the state and the number of interrupts required on all nodes to acquire or release.

2.1 T&T&S Implementation

T&T&S lock first reads the value of the lock variable and issues atomic primitives , i.e., *fetch_and_set*, on the lock variable if value from the read is zero. Otherwise, it backs off and tries the locks again. In our study, we adopt the exponential backoff scheme. When the lock is in local idle state, the lock variable is cached locally and T&T&S implementation performs a local read and a local fetch_and_set operation. T&T&S performs best under this scenario since it requires only two local accesses. When the lock is in remote idle state, the lock variable is cached at a remote node and the first load must traverse up to 4 hops to obtain the shared copy of the lock variable. The subsequent atomic operation must also traverse another 4 hops before the acquire completes. Hence, it takes totally 8 hops even though the lock is free. When the lock is busy, the extra load preceding atomic primitive minimizes the impact of global spinning, but still there is a global flurry of activity at lock release time as each node refetches valid copies of the lock variable. This generates at least 12 global messages to acquire a lock when it is in remote busy state.

2.2 MCS Implementation

MCS locks reduce the amount of global traffic during heavy contention. MCS maintains a distributed waiting queue per lock in shared memory. Data structure for each distributed queue

includes a queue entry per node and a tail pointer. To save remote traffic, the home page for queue entry is allocated at the node that owns the entry. There is a flag field per queue entry that is used for local spinning and signaling. The tail pointer points to the queue entry of the node that is the last one to enter the queue. Acquiring a lock involves only two nodes if lock is currently held: the node in the tail and the one requesting the lock. Releasing lock also involves two nodes: the one releasing the lock and the one next to the releaser in the queue. Unlike the T&T&S, the nodes waiting on a lock in MCS spin locally without generating any global traffic. However, maintaining distributed queue incurs more overhead than T&T&S. Like T&T&S implemetation, MCS avoids generating global traffic when the lock is in the local idle state. When the lock is in remote idle state, MCS needs up to 4 hops for doing *fetch_and_store* into the queue entry at the tail pointer location. When the lock is in remote busy state, requesters enter distributed queue by updating tail pointers. This takes up to 4 hops if valid data is not in home node. It takes another 4 more hops to store its pointer to its own queue entry into its predecessor's queue entry. Then the requester spins locally on the flag in its queue entry waiting for the predecessor's signal. When the predecessor releases the lock, it obtains the pointer to its follower's queue entry in 4 hops, and updates the flag in the queue entry in another 4 hops. Finally, the next node in the queue refetches the valid value of the flag in 4 hops and acquires the lock. That amounts to 12 hops after the lock release. However, MCS restricts the number of global messages and hence performs better than T&T&S when there is heavy contention. Nevertheless, when only a few number of nodes (2 or 3 nodes) contend for the same lock, MCS might perform poorly compared to T&T&S since it requires more global operations to maintain distributed queue.

2.3 Reactive Implementation

Reactive locks adopt to the best behavior of T&T&S or MCS locks based on the observed lock contention. One limitation of Reactive implementation is that they use local information while making the decision to adopt. This may not be precise since it is hard to accurately calculate the degree of lock contention. For example, Reactive implementation may mistakenly switch lock protocol from MCS to T&T&S looking at idle acquires by one of the processes even though that might be followed by a burst of acquire attempts from other processes. After improperly switching, Reactive implementation requires attempts from one node to fail for a few time before the protocol is switched back to MCS.

2.4 MP-LOCK Implementation

MP-LOCK is built upon software and using the existing message passing mechanisms without requiring any special-purpose hardware or atomic primitives. The MP-LOCK model has *lock managers* that manage lock ownerships. We evaluated three lock manager organizations: (i) a single centralized lock manager (MP-cent), (ii) a set of cooperating lock managers running on each node (MP-dist), and an adaptive distributed manager that reverts to centralized mode when there is less contention (MP-react). Rather than using shared memory loads/stores and atomic primitives, MP-LOCK synchronization libraries send lock requests to a lock manager, which will either queue the request until it can be satisfied or forward the request to the lock manager that currently has the lock. We evaluated both user-level and in-kernel lock manager and found that the context switch overhead of user-level lock manager is significant. Therefore, in this paper we restrict our focus to in-kernel lock manager.

A Lock that is in local idle state frequently is often reacquired by the same node before it is acquired by other nodes. Therefore, an efficient lock scheme should "cache" lock ownership locally for subsequent acquires. "Caching" comes naturally for shared memory locks. Therefore, it is important that MP-LOCK implementations have this property. When the lock is in remote busy state and pending lock requests are just queued in centralized lock manager, it takes two hops to transfer ownership after a lock is released (one hop from the lock holder to the lock manager and one hop to the requester). However, if requests are "forwarded" to the node currently holding the lock or to the node at the end of the waiting queue, it costs only one hop to pass lock ownership. When the lock is in remote idle state, it is infrequently reacquired by the same node. This type of lock does not benefit from "caching" and "forwarding". It can be most efficiently handled by returning lock ownership to a centralized manager.

In MP-cent, lock managers handle both lock and unlock requests. A node acquires a lock by performing a non-blocking send to a designated lock manager and then spins while waiting for a reply. The releasing node does a non-blocking send to the central lock manager. Acquire request is granted immediately if the lock is free. Otherwise, acquire request is queued until lock ownership is returned. Conceptually, MP-cent is similar to T&T&S that snoops on a centralized location. However, T&T&S requires up to 8 hops to read valid lock variable and to acquire the exclusive ownerships if the lock is free. Acquiring a lock in MP-cent requires only 2 hops and one interrupt at the central lock manager under all circumstances. MP-cent performs well when there is less contention and different processes acquire the lock. However, the lack of support for "caching" and "forwarding" leads to poor performance when the lock is in local idle or remote busy state.

In MP-dist, lock managers cooperate to manage a distributed queue of pending lock requests. When a process wants to acquire a lock, it consults its local lock manager. If the lock is free and the lock ownership is cached locally, the local lock manager returns the lock immediately. Otherwise, the process sends the lock request to a designated remote lock manager, which either returns the lock or forwards the request. To release a lock, a process consults its local lock manager. If a request is pending, ownership of the lock is forwarded to the requesting process directly in a single message. If no request is pending, the local lock manager caches the lock ownership.

MP-dist performs well for locks that the processes try to acquire when they are heavily contested or frequently reused. During heavy contention, lock ownerhip will be passed between successive owners directly via a single message, and the extra latency and messages required to forward the lock request to the end of the queue is effectively hidden as part of the required stall until the lock is free. When lock reacquisitions (local idle acquire state) is common, MP-dist's caching of lock ownerships makes acquire very cheap. However, when the lock is in remote idle state for majority of acquires, MP-dist performs relatively poorly - it requires on average 3 hops (messages) and 2 interrupts to transfer the ownership from the current holder and to the requester.

Because different applications have very different lock request patterns, we developed a reactive lock protocol, akin to Reactive locks [14], called MP-react. Since lock managers mediate both lock acquire and release requests, they have accurate knowledge of lock access patterns. A central lock manager initially grants "uncachable" locks so that lock ownership is returned when locks are released. Acquire requests on these locks require two hops to be satisfied, as in MP-cent. If the lock manager detects repeated requests from the same node without request from other nodes, it grants a "cachable" instance of the lock so that it can be reacquired by the node without performing any remote operations. When a lock is heavily contested, lock manager will "forward" acquire requests as in MP-dist so that ownership transfers will take one hop. MP-react has better information than Reactive locks because the central lock manager can track global access patterns and thus can make more effective decision about when to switch modes.

We summarize in Table 1 the overhead of acquiring a lock for each of the implementations we considered when the lock is *local idle*, *remote idle*, or *remote busy* state. In the case of *local idle* and *remote idle* states, the acquire overhead shown is the amount of work required to detect that the lock is idle and resume the requesting process. In the case of *remote busy acquire*, acquire overhead is the amount of work required to transfer the lock from a releaser to the acquirer at the release point. When the lock is in local idle state, only local operations are required for all lock schemes except MP-cent to acquire a lock. In both remote idle and busy states, shared memory locks always require more hops than MP-LOCK. However, the MP-LOCK implementations might

Lock Scheme	Local Idle	Remote Idle	Remote Busy
T&T&S	0h	8h	12h
MCS	0h	4h	12h
Reactive	0h	4h	12h
MP-cent	N/A	2h+1i	2h+1i
MP-dist	0h	3h+2i	1h
MP-react	0h	2h+1i	1h

Table 1 Cost and Latency of Various Lock Schemes (h: network hop. i: interrupt)

incur interrupts due to their software implementation. We will describe efforts to minimize the overhead of MP-LOCK's software implementation in Section 3.

3 Implementation Issues in MP-LOCK

To minimize the overhead of MP-LOCK's software implementation, we used a low latency message passing mechanism called *Direct Deposit (DD)* [26] and embedded lock managers in the kernel. We also carefully distributed lock management across nodes to minimizeload imbalance effects. In this section, we describe the pertinent implementation details that impact performance.

To avoid load imbalance, we employ one lock manager per node for all three MP-LOCK algorithms. We statically assign each lock to a single lock manager in a round robin fashion when it is allocated (e.g., lock 1 is managed by the lock manager on node 1, lock 2 by node 2,..., and lock N+1 by node 1). Other distribution methods, e.g., first touch, are possible but were not considered. This distribution of lock management mitigated the effect load imbalance, as shown in Section 5.

To alleviate the overall impact of using software lock managers, we embed them in the kernel. This reduces the impact to application processes running on the node of handling remote requests. The decision to embed the lock managers in the kernel raises the question of where to cache lock ownership in the MP-dist and MP-react algorithms when the lock is released and no other request is pending: in the application process or in the kernel lock manager. Caching in the user application allows a lock to be reacquired by the same process inside of a local library routine, making it effectively free. However, to satisfy a remote request, we would be forced to deliver the application process a signal to reclaim the lock ownership, which takes 3500 cycles to perform on our kernel. Therefore, we chose to cache lock ownership within the kernel lock managers. Application processes use a light weight system call to return lock ownership to the kernel-embedded lock managers, which allows remote lock requests to be satisfied quickly without performing a context switch, and to reacquire locally cached locks.

Finally, we employed the low latency Direct Deposit (DD) protocol to communicate between lock managers. DD is a sender-based protocol (SBP) [28], which means that it employs a connection-based mechanism that enables the sender to manage a reserved receive buffer within the receiving process' address space that is obtained when the connection is established. The sender directs placement of messages within that buffer via an offset carried within the message header. The semantics of DD allow for *asynchronous* sends, i.e., the send call can simply request transmission of the message and return immediately. Coupled with an appropriate network interface, an SBP can achieve 0-copy message reception directly to the receiver's virtual address space and allow transmissions to occur in parallel with continued computation by the sending process.

DD supports user mode message reception. A user level receive consists simply of checking the valid flag of the incoming connection's notification. Polling for incoming messages is thus an extremely inexpensive operation. The message buffers can completely reside in the receivers' address space, which yields a used-mode receive capability.

4 Performance Evaluation

4.1 Experimental Setup

All experiments were performed using an execution-driven simulation of the HP PA-RISC architecture called Paint (PA-interpreter)[25]. Paint was derived from the Mint simulator[29]. Our simulation environment includes detailed simulation modules for a first level cache, system bus, memory controller, network interconnect, and DSM engine. It provides a multiprogrammed process model with support for operating system code, so the effects of OS/user code interactions are modeled. The simulation environment includes a kernel based on 4.4BSD that provides scheduling, interrupt handling, memory management, and limited system call capabilities. Simulating the kernel provides a fair accounting for our software-based MP-LOCK overheads.

All experiments were conducted using a Simple-COMA-based architecture [22] with an infinite DRAM page pool and a 1-Mbyte direct-mapped L1 cache. The reason we chose a large L1 cache size and a "perfect" Simple-COMA-based architecture is to eliminate the effects of shared data conflict/capacity misses, which could lead to skewed performace for the T&T&S and MCS algorithms shared due to memory allocation effects. The modeled processor and DSM engine are clocked at 120MHz. The system bus modeled is HP's Runway bus, which is also clocked at 120MHz. All cycle counts reported herein are with respect to this clock. We model a 4-bank main memory controller that can supply data from local memory in 58 cycles. For our interconnect, we modeled a Myrinet network [2] with 1-cyle propagation delays and a 16-port Myrinet switch. For experiments with

fewer than 16 nodes, only one switch is required. For the 32-node configuration, we use a 2X2 mesh topology. We ran our experiments with two different switch fall through delays of 4 and 176 cycles to model high-end commercial DSM systems such as the SUN UE10000 [27], SGI Origin 2000 [13] and Mercury Interconnect Architecture [30] that use specialized high speed interconnects and other DSM systems or clusters of workstations that use less aggressive off-the-shelf interconnects [15]. The characteristics of the L1 cache and network that we modeled are shown in Table 2. As a result of the modeled machine characteristics, the average 2-hop lock acquire operation required 1400 to 1700 cycles in the MP-LOCK implementations depending on the network delay. Finally, we varied the number of nodes from 4 to 32.

4.2 Benchmark Programs

We used five programs to conduct our study: `mp3d` from the SPLASH benchmark suite [24], `barnes`, `radiosity`, and `raytrace` from the SPLASH-2 benchmark suite [31], and `spark98` from a sparse matrix kernel suite [18]. Table 3 shows the inputs used for each test program.

Table 4 presents the distribution of lock categories for the applications we studied. Because the access patterns do not change dramatically with different interconnect speeds, we show the distribution only for the fast interconnect model.

Component	Characteristics
L1 Cache	Size: 1-Mbytes. 32 byte lines, direct-mapped, virtually indexed, physically tagged, non-blocking, up to one outstanding miss, write back, 1-cycle hit latency
Networks	1 cycle propagation, 16-port switch , port contention (only) modeled Fall through delay: 4 and 176 cycles

Table 2 Cache and Network Characteristics

Applications	Type of Simulation	Inputs
barnes	Barnes-Hut N-body	16k particles
mp3d	Hypersonic flow	20k molecules, 10 steps, the lock version
radiosity	The equilibrium distribution of light in a scene	room
raytrace	3-D rendering	car
spark98	Kernel doing sparse matrix vector product operations	sf10 sparse matrix, the lock version

Table 3 Classification of the simulated benchmarks and their inputs

Applications	total lock pairs (locks per milisec)	Local Idle	Remote Idle	Remote Busy (interrupts inside CS)
barnes (4 nodes)	68871(15.0)	73%	25%	2% (2%)
barnes (8 nodes)	68972(31.0)	63%	30%	8% (4%)
barnes (16 nodes)	69150(60.0)	52%	29%	19% (6%)
barnes (32 nodes)	69468 (114.0)	47%	28%	25% (7%)
mp3d (4 nodes)	407996(148.0)	26%	73%	1% (0%)
mp3d (8 nodes)	407918(235.0)	13%	86%	0% (0%)
mp3d (16 nodes)	408020(403.0)	7%	92%	1% (0%)
mp3d (32 nodes)	408126(700.0)	4%	94%	2% (0%)
radiosity (4 nodes)	198991(23.0)	57%	37%	6% (3%)
radiosity (8 nodes)	210186(42.0)	47%	33%	20% (6%)
radiosity (16 nodes)	266657(84.0)	34%	32%	34% (3%)
radiosity (32 nodes)	474261(98.0)	19%	22%	59% (2%)
raytrace (4 nodes)	95472 (26.0)	38%	52%	10% (2%)
raytrace (8 nodes)	95480(46.0)	20%	43%	37% (5%)
raytrace (16 nodes)	95497 (79.0)	5%	16%	79% (4%)
raytrace (32 nodes)	95532 (105.0)	1%	3%	96% (0%)
spark98 (4 nodes)	2088644(290.0)	77%	23%	0% (0%)
spark98 (8 nodes)	2088648(505.0)	75%	25%	0% (0%)
spark98 (16 nodes)	2088656(849.2)	69%	31%	0% (0%)
spark98 (32 nodes)	2088672(1345.0)	63%	37%	0% (0%)

Table 4 Lock access patterns in `barnes`, `mp3d`, `radiosity`, `raytrace` and `spark98`. Results are obtained from the MP-dist simulations with the fast interconnect.

We believe various synchronization access patterns are well represented by the choice of these five applications. Among these five applications, lock contention in three applications goes up, although to different degrees, as the number of nodes increases. In `barnes`, 50% of locks are reused even with 32 nodes due to the application's excellent temporal locality. However, the number of remote busy locks increases from 2% to 25%. Most local idle locks in `radiosity` are transformed into remote busy locks as the number of nodes increases. In an extreme case, 96% of locks fall into the remote busy category in `raytrace` with 32 nodes. `Mp3d` and `spark98` have the lowest level of lock contention among the five applications. 99% of lock acquires in these two applications are idle locks. The degree of lock contention remains the same even as the number of nodes varies. Because of `mp3d`'s poor shared data temporal locality, most of its locks that protect the shared data fall into the remote idle category. As opposed to `mp3d`, locks in `spark98` show excellent temporal locality and most of them can be satisfied locally if lock caching is implemented. In Table 4, `interrupts inside CS` represents the frequency with which nodes executing inside a critical section are interrupted by incoming forwarded requests, which only occurs in MP-dist and MP-react. When this occurs,

critical sections are artificially lengthened, which degrades performance. Fortunately, it does not occur frequently in the applications we studied.

5 Results

Figures 1, 2, and 3 present the performance of MCS, T&T&S, MP-cent, MP-dist, and MP-react on these five applications. Four graphs are presented for each application. The top two graphs present the results using the fast interconnect model, while the bottom two graphs present the results using the slow interconnect model. The two graphs on the left present the execution time of the application using each lock implementations relative to the MCS lock version. The two graphs on the right break down the time the application spent performing various tasks. *U-shmem* denotes cycles spent while accessing shared memory – the increased DSM controller occupancy of shared memory locks can cause higher shared memory stall times. *Kern* denotes cycles spent performing the basic kernel operations required by all configurations (e.g., system calls) plus time spent executing the kernel-embedded lock managers in MP-LOCKs. *U-instr* and *U-lclmem* denotes cycles spent performing user-level instructions and accessing non-shared (local) memory. *Barrier* and *lock* denote cycles spent waiting for barrier and lock/unlock operations to complete, respectively. All results include only the parallel phase of the programs.

The five applications can be divided into two categories: applications whose lock access pattern changes with the system configuration (`barnes`, `radiosity` and `raytrace`) and applications whose lock access patterns remain unchanged (`mp3d` and `spark98`).

Locks in `barnes` are used as follows. Accesses to each space cell are protected by per-cell locks, and the global maximum and minimum values also are protected by locks. During each time step, processes load bodies into an octree structure that represents 3-D space. This phase generates the majority of the program's synchronization operations. Although the average critical section is long, the chance of two processes contending for the same lock is small. As shown in Table 4, only 2% of the locks are busy in a 4-node system and 73% of the locks are "reused" by the same node. Octree initialization is followed by a long computation phase, which accounts for most of the execution time. Thus, as detailed in Figure 1, the time spent on synchronization is small compared to the overall execution time, so there is little need for special hardware support for locking – MP-LOCKs can provide equal or better performance than shared memory locks. As shown in Figure 1, the shared memory lock implementations (MCS and T&T&S) and MP-LOCK implementations (distributed and reactive) perform equally well in all cases except for the 32-node

slow interconnect configuration. The lack of caching penalizes MP-cent, which underperforms the other four implementations by about 5%.

A number of patterns are evident in the results. The degree of lock contention increases from 2% to 25% as the number of nodes increases from 4 to 32 (see Table 4). The relatively high degree of lock contention in the 32-node configuration causes the performance of T&T&S to drop dramatically, which is in line with previous studies [10, 14, 16]. In addition, the performance of shared memory locks is heavily impacted by interconnect latency and the number of remote memory accesses required. Even MCS requires more remote memory accesses than MP-dist as shown in Table 1, and as a result MP-dist outperforms MCS by 8% in the 32-node slow interconnect configuration. Finally, increased controller occupancy and cache conflicts in large configurations increases the user shared memory access time noticeably for the shared memory locks, while the lock manager overhead of the MP-LOCK implementation scales well.

The `radiosity` program is used to produce realistic computer-generated images of complex scenes by accounting for both direct illumination by light sources and indirect illumination through multiple reflections. Locks are used to protect a number of data structures. First, load balancing is implemented using distributed task queues – idle processes dequeue tasks from non-empty task queues maintained by other processes, which are protected by locks. Another lock isused to implement a global barrier and another is used to protect a buffer pool. The degree of contention for these three sets of locks depends on the number of nodes. For smaller systems (4 - 8 nodes), most of the time these locks fall into the local idle category, but as the number of nodes increases, contention for these locks increases. In addition, locks are used to protect patches that make up the image as they are subdivived, but there is little contention on these locks due to their fine granularity. However, the poor temporal locality of these patch locks result in most accesses to them being of remote idle variety – in the 4-node configuration, one third of lock accesses are to remote idle locks.

Due to the low level of contention and MP-LOCK's higher software overhead, MCS locks perform 18% better than MP-LOCKs in the 4-node fast interconnect configuration. However, with a slow interconnect, the increased shared memory access time reduces the performance gap to 13%. As the number of nodes increases, contention increases. This leads to an decrease in the percentage of local idle locks from 57% to less than 20%, and an increase in the percentage of remote busy locks from 6% to about 60%. Regardless of configuration, the percentage of remote idle locks is constant about 20% because of the large amount of fine grained patch locks, which have poor locality. As a result of these changed in lock access pattern, MP-react starts to outperform MCS at the 16-node configuration and by the time the configuration reached 32 nodes, it outperforms MCS

locks by up to 64%. Comparing just the MP-LOCK schemes, when contention is low, MP-cent outperforms MP-dist, as expected, but as the number of nodes and/or network latency increases, MP-dist prevails. MP-react, which is able to adapt to the best of MP-cent and MP-dist, performs up to 10% better than either schemes.

`Raytrace` renders a three-dimensional scene using ray tracing. Major data structures include the ray trees, a hierarchical uniform grid, and an octree-like data structure that represents the scene being rendered. Locks in `raytrace` are used as follows. All shared data is allocated from a pre-allocated shared memory pool protected by a single lock. Like `radiosity`, `raytrace` uses distributed task queues, with a lock protecting each queue. In the 4- and 8-node configurations, the lock protecting the memory pool falls is mostly remote idle, a category that accounts for 52% of lock requests. In these same small configurations, the locks protecting the distributed task queues are mostly local idle. Thus, for small configurations, shared memory locks perform up to 14% better than MP-LOCKs. As number of nodes increase and load imbalance occurs, processes begin to perform task stealing, which causes the task queue locks to become busy. The memory pool lock also becomes busy. As a result, 97% of the locks in the 32-node configuration are heavily contested, which causes MP-LOCKs to perform up to 75% for several reasons. First, MP-LOCKs can better handle highly contested locks because they can forward locks in a single message. Second, the impact of lock manager interrupts on user processes is amortized effectively. Finally, MP-LOCKs do not increase DSM controller occupancy, which helps to reduce shared memory access times.

`Mp3d` and `Spark98` represent applications whose lock contention does not change dramatically with system configuration. `Mp3d` solves a problem in rarefied fluid flow simulation. Most synchronization in `mp3ed` occurs during the `move` phase. Locks are used to atomically update the cell data of the active space array. The degree of contention on these locks is extremely low – as presented in Table 4, 2% or fewer of the locks are found in busy state. Locks in `mp3d` have poor temporal locality and there is very little reuse, so most locks fall into the remote idle category. In this case, MP-LOCKs require up to two interrupts (one at the lock manager and one at the current lock holder) and three interconnect hops to acquire a lock. Thus, for 4 and 8 nodes, shared memory locks outperform the MP-LOCK implementations by up to 180%. The performance gap shrinks as the number of nodes and/or network latency increases. For example, MP-LOCKs perform better than MCS in the 16-node configuration with a slower network and up to 186% better in the 32-node configuration. Performance improves even though the number of locks in the remote idle state increases from 73% in the 4-node configuration to 94% in the 32-node configuration. Two factors can explain the poor scalability of shared memory locks: (i) the tight dependence between shared memory lock performance and remote memory access latency and (ii) the severe impact on

DSM controller occupancy caused by shared memory locks. Comparing just the MP-LOCK mechanisms, MP-cent outperforms MP-dist in the 32-node configuration because 94% of locks accesses are to remote idle locks. However, only 73% of lock accesses are to remote idle locks in the 4-noode configuration, so MP-dist outperforms MP-cent because of lock caching.

`Spark98` is a sparse matrix multiplication kernel that performs a sequence of sparse matrix vector product (SMVP) operations. Each element in the result vector is protected by a lock. After multiplying a row of the sparse matrix times the dense vector, a process locks the result vector elements for which it computed a non-zero inner product so that it can add its partial result to the result vector. These locks have good temporal locality due to the way that processes are assigned work. The number of locks that are reused ranges from 77% in the 4-node configuration to 63% in the 32-node configuration. Hence, `spark98` benefits greatly from lock caching, as can be seen by the relative performance of MP-cent and MP-dist. In the 4-node and 8-node configurations, shared memory locks perform 70% better than MP-LOCKs due to their low-latency lock/unlock routines. With increase in the number of nodes and/or network latency, however, MP-LOCKs perform up to 163% better than shared memory locks. Because of high percentage of reuse, MP-dist outperforms MP-cent. However, as the number of nodes increases from 4 to 32, the number of locks in remote idle state increases from 23% to 37%, which dramatically closes the performance gap between MP-cent and MP-dist.

In summary, for applications with high lock contention, the best MP-LOCK algorithm outperforms the best shared memory lock algorithm by up to 186%. In particular, MP-LOCKs tend to outperform shared memory locks once the the system size reaches 16 nodes with a fast interconnect or 8 nodes with a slow interconnect. The superior scalability of MP-LOCKs on these applications occurs for several reasons. First, MP-LOCKs handle remote busy locks better than shared memory locks, because it can forward lock ownership in a single message. To achieve similar performance, shared memory locks would require special hardware shared memory protocols not present in modern machines [17]. Second, MP-LOCKs neither increase DSM controller occupancy nor interfere with shared memory data accesses, which can lead to significantly lower average remote memory latency for non-lock shared data. Finally, the software overhead induced by lock managers can be amortized across nodes, which reduces its impact. For applications with low lock contention, MP-LOCKs underperform the best shared memory lock implementation on small systems (4 nodes or 8 nodes) by up to 15% in three applications and no more than 180% in the remaining two applications. Fast hardware shared memory lock implementations can handle low contention locks more efficiently than MP-LOCKs. Given the trends we observed, we expect that MP-LOCKs will scale better than shared memory locks as the number of nodes increases beyond 32 nodes. Thus,

we believe that MP-LOCKs are an attractive alternative to hardware synchronization primitives for future scalable shared memory multiprocessors that support efficient message passing.

6 Related Work

In addition to their Reactive lock mechanism that adapts between T&T&S and MCS semantics, Lim and Agarwal [14] proposed a reactive lock mechanism that adapts to either shared memory or message-based style locking. They found that a message-passing centralized queue-based lock (MPCQL) starts to ourperform T&T&S as the number of nodes exceeds four, which agrees with our results. However, they found that MCS locks consistently outperformed MPCQL. Their study was limited to a set of microbenchmarks, rather than whole programs as presented here, and they considered only a single centralized message-passing-based protocol for one set of interconnect speeds and remote shared memory latencies. Our results show that a more adaptive MP-LOCK protocol can outperform even MCS locks, depending on the application locking pattern, network latency, and machine size.

Kagi and Goodman proposed a software version of QOLB, called SOFTQOLB [10]. SOFTQOLB's implementation is based on the Tempest interface [20]. When they compared the performance of SOFTQOLB against MCS locks and a centralized queue-based lock mechanism, they found that message-passing locks can be as efficient as shared memory locks at low lock contention and can outperform them when lock contention is high. However, their study only considered microbenchmarks and low end clusters of workstations, which makes it difficult to compare their results to ours directly. They did not attempt to identify the performance bottlenecks of their SOFTQOLB implementation, nor did they suggest ways to exploit software's inherent flexibility. Nevertheless, like our proposed MP-LOCK mechanisms, SOFTQOLB provides an efficient alternative to conventional hardware locks for emerging scalable multiprocessors.

7 Conclusions

In this paper, we demonstrate that software-based locks are an attractive alternative to hardware-based implementations. The so-called MP-LOCK approach is based on efficient message passing mechanisms that can be supported by most contemporary multiprocessor interconnects. By basing locks on message passing rather than dedicated hardware, MP-LOCK reduces the design complexity and runtime occupancy of DSM controllers. In addition, MP-LOCKs can exploit software's inherent

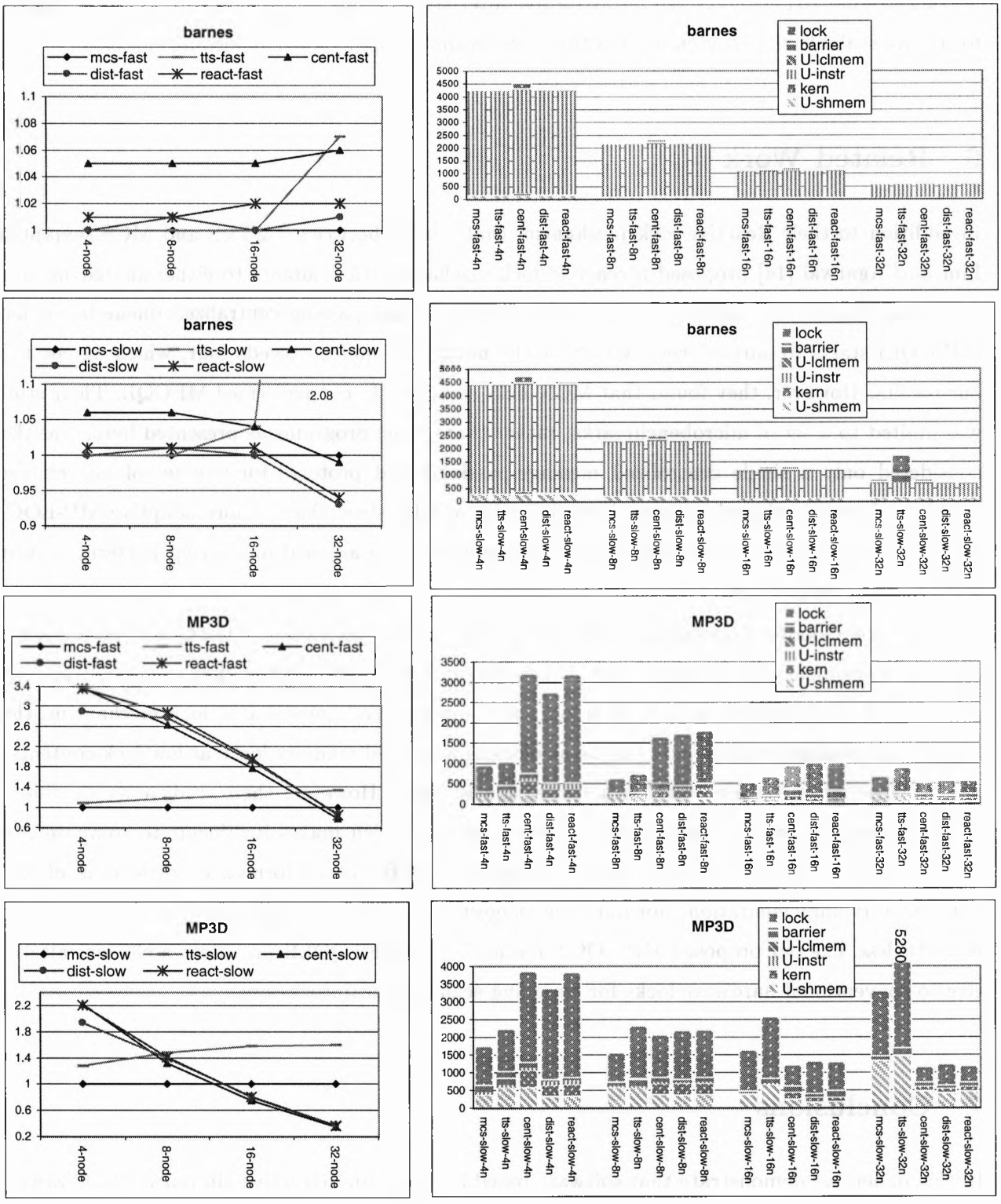

Figure 1 Performance Charts for `barnes` and `mp3d`. *Left Column*: Execution Time Relative to MCS. *Right Column*: Breakdown Execution Time (msecs).

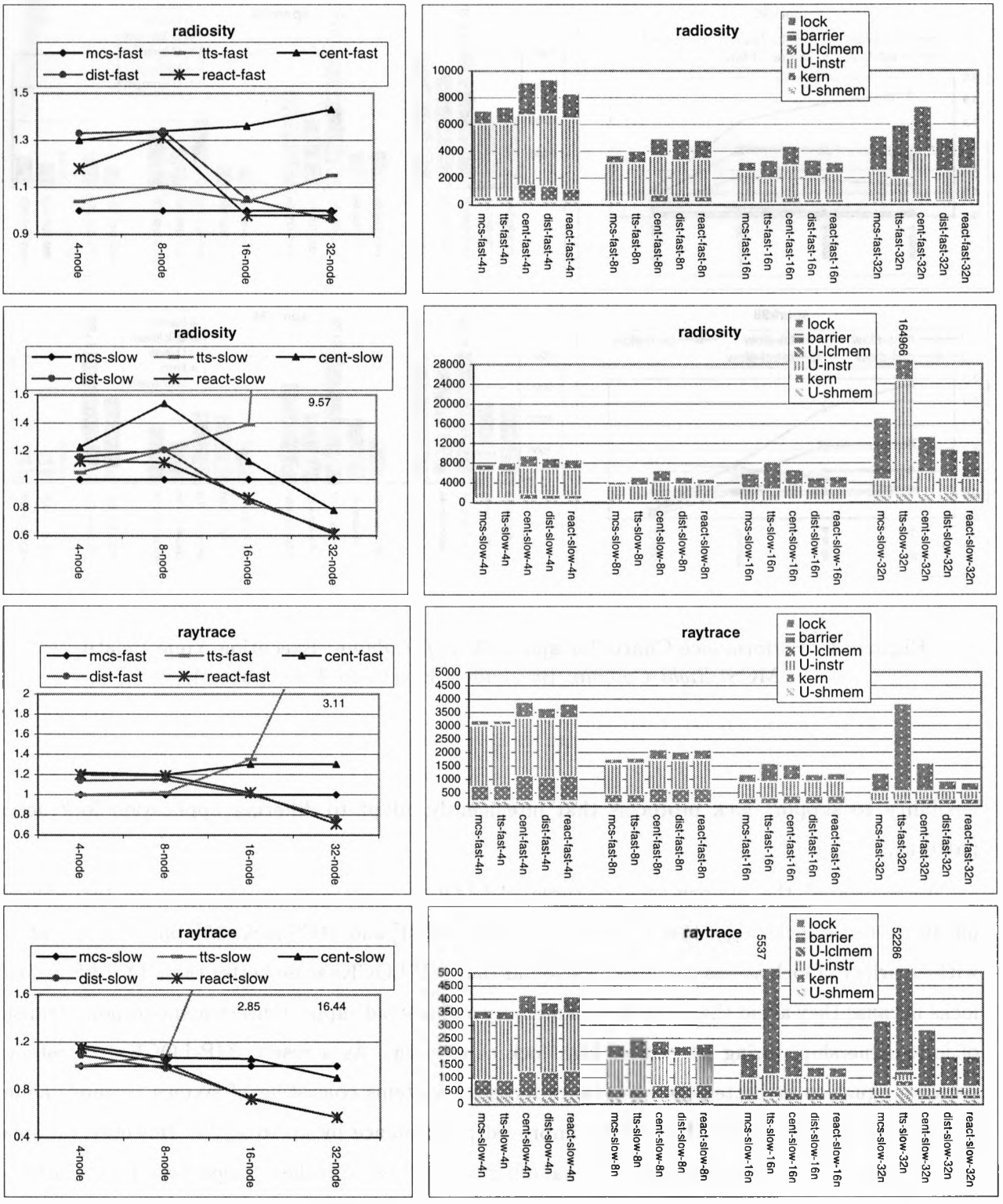

Figure 2 Performance Charts for `radiosity` and `raytrace`. *Left Column*: Execution Time Relative to MCS. *Right Column*: Breakdown Execution Time (msecs).

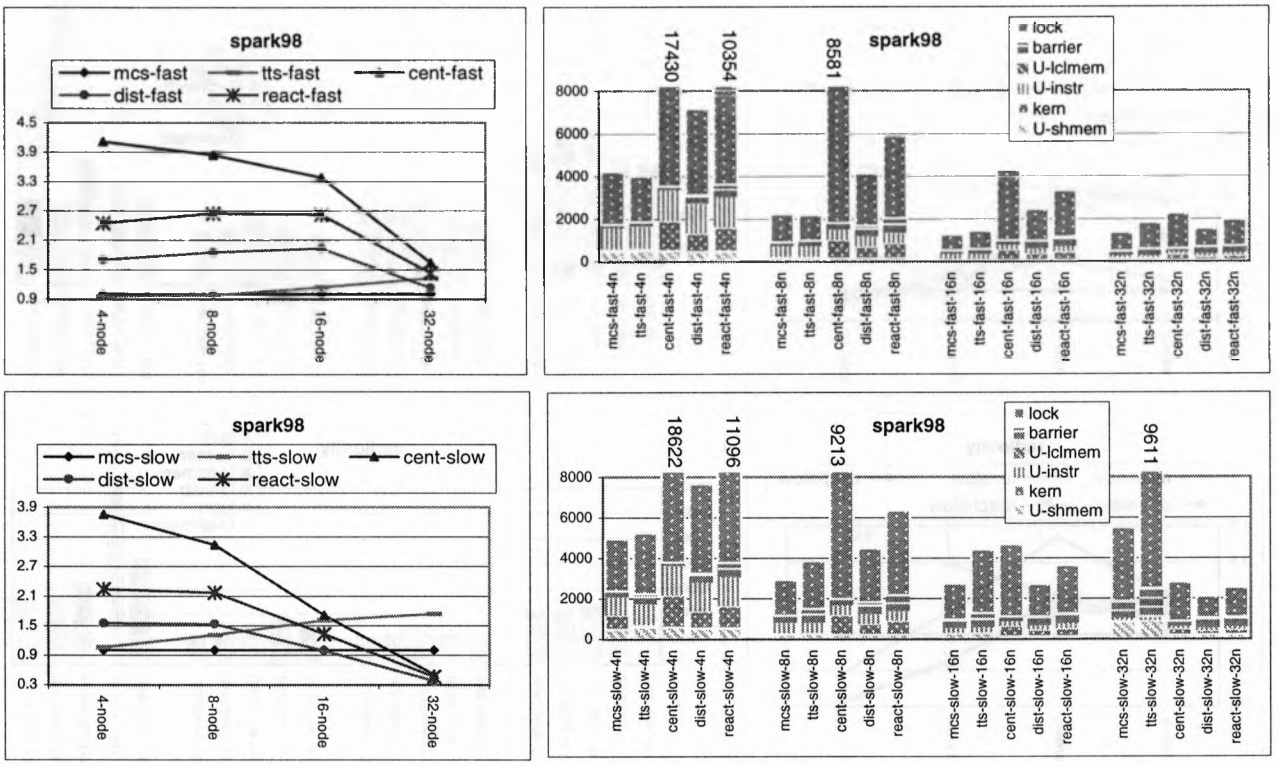

Figure 3 Performance Charts for `spark98`. *Left Column*: Execution Time Relative to MCS. *Right Column*: Breakdown Execution Time (msecs).

flexibility to support lock protocols that intelligently adapt to differing application lock access patterns.

We evaluated the performance of three MP-LOCK algorithms against that of two efficient hardware-based locks algorithms, test-and-test-and-set[21] and MCS locks[16], on five applications with a variety of lock access patterns. We found that MP-LOCKs scale better than T&T&S or MCS locks because they avoid the use of shared memory and instead support direct point-to-point transfer of lock ownership during periods of high lock contention. As a result, MP-LOCKs consistently perform equal to or better than hardware locks for systems consisting of sixteen or more nodes. In the extreme, the use of MP-LOCKs improved performance by up to 186%. However, for small system sizes, e.g., 4 - 8 nodes, interrupt handling and software overhead caused the performance of the MP-LOCK versions to lag that of shared memory locks. However, the difference was no more than 18% in three applications and no more than 180% in the remaining two applications.

Focusing on the MP-LOCK algorithms in isolation, we found that MP-cent performed best for applications like `mp3d` with poor lock locality, and thus frequent remote idle accesses. The

reason is that for these applications relinquishing locks back to a per-lock centralized lock manager minimizes message traffic. However, when contention is high or locks are reused frequently, MP-dist significantly outperforms MP-cent, because direct lock forwarding and lock caching effictively handle these situations. MP-react exploits global access pattern observations to adaptively switch between centralized and distributed modes, which leads to good overall performance and the best performance for applications that demonstrate a mix of access patterns.

This paper makes several contributions. We present the results of the first study that compares the performance of message passing locks and shared memory locks on macrobenchmarks. We took great pains to conduct a fair comparison by including a detailed 4.4BSD-based kernel in our simulation environment. This kernel provides scheduling, interrupt handling, and system call capabilities to accurately simulate the software overhead of the proposed message passing mechanisms. Second, we identified the tradeoffs for shared memory locks and message passing locks as system sizes and network latencies vary. These results should assist future architects when designing their synchronization mechanisms. Third, we classified the lock access patterns of five well-known shared memory benchmarks on various number of processors, which will help other researchers understand the locking behavior of these applications. Finally, we provided guidelines for designing synchronization mechanisms in clusters of workstations that are equipped with message passing communication mechanisms. For example, we show that lock caching is essential when designing message-passing based locks.

In the future, we plan to further minimize interrupt overhead, which causes performance problems in small systems, by evaluating various application-level polling strategies that will allow us to eliminate the need for kernel-level lock managers. We also plan to investigate techniques to exploit QOLB-style lock-data collocation in the MP-LOCK algorithms. Doing so has the potential to eliminate a large amount of coherence traffic. Since messages can easily carry the data protected by a lock along with lock ownership, this might appear trivial at first glance. However, a straightforward implementation would require modification to conventional DSM controllers to avoid coherence traffic in response to writing the data to the new lock owner's memory. Finally, we plan to investigate more intelligent adaptive locking protocols that better exploit the global lock access pattern information that can be gleaned by lock managers.

References

[1] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[2] 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(1):29–36, February 1995.

[3] G. Buzzard, D.Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, October 1996.

[4] R.P. Case and A. Padegs. Architecture of the ibm system 370. *Communications of the ACM*, 21(1):73–96, January 1978.

[5] J. R. Goodman, M. K. Vernon, and P.J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[6] A. Gottlieb and C.P. Kruskal. Coordinating parallel processors: A parallel unification. *Computer Architecture News*, 9(6):16–24, October 1981.

[7] International Business Machines Inc. *IBM System/360 Principles of Operation*, ninth edition, May 1970.

[8] D.V. James. Distributed directory scheme: Scalable coherent interface. *IEEE Computer*, 23(6):74–77, June 1990.

[9] E.H. Jensen, G.W. Hagensen, and J.M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Lab, November 1987.

[10] A. Kagi, D. Burger, and J.R. Goodman. Efficient synchronization: Let them eat qolb. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[11] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message-passing and shared-memory; early experience. In *Proceedings of the 1993 Conference on the Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[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] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *ISCA97*, pages 241–251, June 1997.

[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] T. Lovett and R. Clapp. STiNG: A CC-NUMA compute system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[16] 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.

[17] 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.

[18] D. O'Hallaron, J. Shewchuk, and T. Gross. Architectural implications of a family of irregular computations. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 80–89, February 1998.

[19] S. Paikin, Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (fm) for myrinet. In *Proceedings of Supercomputing '88*, 1995.

[20] 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.

[21] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, May 1984.

[22] 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.

[23] Steven Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[24] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[25] L.B. Stoller, R. Kuramkote, and M.R. Swanson. PAINT: PA instruction set interpreter. Technical Report UUCS-96-009, University of Utah - Computer Science Department, September 1996.

[26] L.B. Stoller and M.R. Swanson. Direct deposit: A basic user-level protocol for carpet clusters. Technical Report UUCS-95-003, University of Utah - Computer Science Department, March 1995. Also available via WWW under http://www.cs.utah.edu/projects/avalanche.

[27] Sun Microsystems. Ultra Enterprise 10000 System Overview. http://www.sun.com/servers/datacenter/products/starfire.

[28] M.R. Swanson and L.B. Stoller. Low latency workstation cluster communications using sender-based protocols - computer science department. Technical Report UUCS-96-001, University of Utah, March 1996. Also available via WWW under http://www.cs.utah.edu/projects/avalanche.

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

[30] W. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. The mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *ISCA97*, June 1997.

[31] 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.