MP-LOCKs: Replacing H/W Synchronization Primitives with Message Passing

Chen-Chi Kuo, John Carter, and Ravindra Kuramkote*

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

Abstract

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 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 implementation guidelines. 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), three applications with MP-LOCKs slow down by no more than 18%, while the other two slowed by no more than 180% due to higher software overhead. 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.*

1 Introduction

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*. 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*). Recently, however, the designers of the Cray T3E broke this rule and abandoned the dedicated high performance hardware barrier network that was supported in the T3D. 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 spin on shared memory locations using hardware T&S instructions. 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 [13] add an extra shared-memory "load" before the T&S primitive to eliminate these unnecessary migrations. However, their performance degrades when multiple nodes are waiting for a lock to be released

Mellor-Crummey and Scott (MCS) [9] locks avoid global spinning entirely by maintaining a distributed queue of processes waiting for a lock. This design can improve performance by reducing global communication and it 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 [8] 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 [4] associates a special lock bit with cache lines of data. 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. Because they are unsuitable for implementation on multiprocessors based on current commodity microprocessors, we do not consider QOLB locks further.

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.

Many contemporary multiprocessor architectures support efficient message passing in addition to shared memory [5, 7, 12], and an increasing number of high performance of network interfaces and protocols have been proposed [1, 2, 11, 15]. We propose that low latency message passing 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 synchroniza-

Email addresses: {chenchi,retrac,kuramkot}@cs.utah.edu.
This research was supported in part by the Space and Naval Warfare Systems Command (SPAWAR) and the Advanced Research Projects Agency (ARPA), under SPAWAR contract No.#N0039-95-C-0018 and ARPA Order No.#B990. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, SPAWAR, or the US Government.

tion 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 [4].

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. Section 2 presents three MP-LOCK algorithms. We describe our simulation environment, and test applications in Section 3, and present the results of our experiments in Section 4. Finally, we draw conclusions in Section 5.

2 Design of MP-LOCK

MP-LOCK is a software-based lock mechanism that requires no special-purpose hardware or atomic primitives. The MP-LOCK model has *lock managers* that manage lock ownership. 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 (iii) an adaptive distributed manager that reverts to centralized mode when there is little contention (MP-react). Rather than using shared memory loads/stores and atomic primitives, MP-LOCK synchronization libraries send lock requests to a lock manager. For each request, the lock manager will either queue the request until it can be satisfied or forward it to the lock manager that currently has the lock. We evaluated both user-level and in-kernel lock managers and found that the context switching overhead of user-level lock managers is significant. Therefore, in this paper we restrict our focus to in-kernel lock managers.

To facilitate designing MP-LOCK algorithms, we classify lock acquires into three categories depending on the state the lock is in when the acquire is initiated: *remote idle acquire*, *local idle acquire*, and *remote busy acquire*. A *remote idle acquire* occurs when some node Y attempts to acquire the lock after some other node X has released it. The acquire in this case will succeed immediately after the remote operations required to detect and update the lock status complete. A *local idle acquire* is the same as a *remote idle acquire* state, except X and Y are the same nodes. Both of the idle cases occur frequently when there is little contention. A *remote busy acquire* occurs when an 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 (which we call *hops*) that must be sent to acquire or release a lock and the number of interrupts required on all nodes to acquire or release.

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. A releasing node also does a non-blocking send to the central lock manager. An acquire request is granted immediately if the lock is free. Otherwise, the acquire request is queued until lock ownership is returned. Conceptually, MP-cent is similar to T&T&S, which snoops on a global location. However, T&T&S requires up to 8 hops to read a valid lock variable and to acquire exclusive ownership when 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 multiple processes acquire the lock with little contention. However, the lack of support for "caching" and "forwarding" leads to poor performance for local idle or remote busy acquires.

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 are heavily contested or frequently reused. During periods of heavy contention, lock ownership 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 reacquisition (local idle acquires) is common, MP-dist's caching of lock ownership makes acquires very cheap. However, when most acquires are remote idle 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 [8], 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 intervening requests 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, the lock manager will "forward"

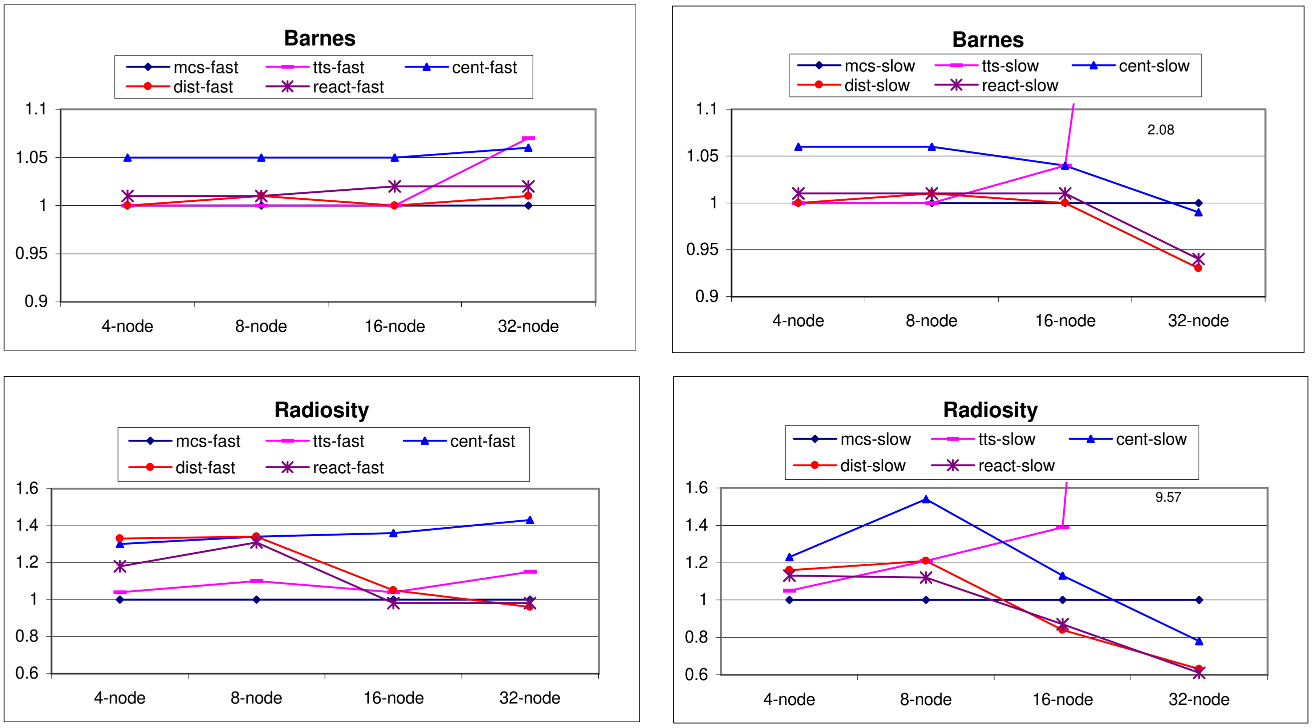

Figure 1. Performance Charts for `barnes` and `radiosity` (execution time relative to MCS).

acquire requests as in MP-dist so that ownership transfers will take one hop. MP-react has better information about access patterns than Reactive locks because the central lock manager can track all accesses. Thus, MP-react can make more effective decisions about when to switch modes.

To minimize the overhead of MP-LOCK's software implementation, we used a low latency message passing mechanism called *Direct Deposit (DD)* [15] and embedded lock managers in the kernel. We also carefully distributed lock management across nodes to minimize load imbalance effects. The pertinent implementation details can be found in an expanded version of this paper [6]

3 Simulation Methodology and Benchmarks

All experiments were performed using an execution-driven simulation of the Avalanche architecture [3]. 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. For our interconnect, we modeled a Myrinet network. We ran our experiments with two different switch fall through delays of 4 and 176 cycles to model both high-end commercial DSM systems and DSM systems or clusters of workstations that use less aggressive off-the-shelf interconnects.

We used five programs to conduct our study: `mp3d` from the SPLASH benchmark suite [14], `barnes`, `radiosity`, and `raytrace` from the SPLASH-2 benchmark suite [16], and `spark98` from a sparse matrix kernel suite [10]. We believe a broad set synchronization access patterns are well represented by these five applications.

The detailed simulation parameters, application sizes and the distribution of lock categories for the applications we studied can be found in the expanded version of this paper [6].

4 Results

Figures 1 and 2 present the performance of MCS, T&T&S, MP-cent, MP-dist, and MP-react relative to the MCS lock version on three of the five applications we studied. We present two graphs for each application. The graph on the left presents the results using the fast interconnect and the graphs on the right present the results using the slow interconnect. Graphs for `raytrace` and `spark98` and detailed breakdown graphs of the execution time can be found in the expanded paper [6].

In `barnes`, most of synchronization occurs while initializing an octree structure that represents 3-D space. Although the average critical section is long, the chance of two processes contending for the same lock is small. Therefore, only 2% of the lock acquires occur when the lock is busy and 73% of the locks are "reused" by the same node in a 4-node system. Since the time spent on synchronization is small compared to the overall execution time, 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%. As the number of nodes increases from 4 to 32, the degree of lock contention increases from 2% to 25%. The controller occupancy and cache conflicts also increase the user shared memory access time noticeably for the shared memory locks. As a result, MP-dist outperforms MCS by 8% at the 32-node slow interconnect configuration.

In `radiosity`, the degree of contention for locks used to pro-

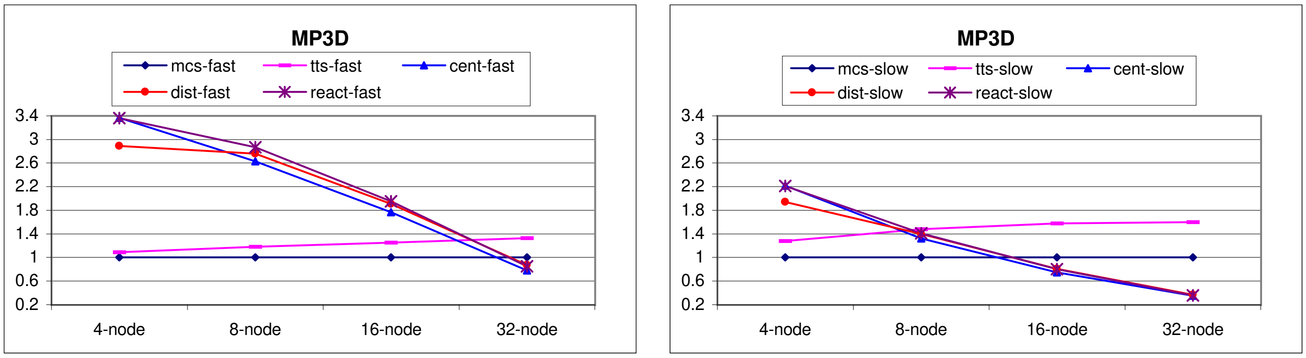

Figure 2. Performance Charts for `mp3d` (execution time relative to MCS).

tect distributed queues, a global barrier, and a buffer pool depends on the number of nodes. For small systems (4 - 8 nodes), most lock acquires are local idle acquires. Locks used to protect image patches show little contention due to their fine granularity. However, the poor temporal locality of these patch locks result in most accesses to them being of the remote idle variety. 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 acquires from 57% to less than 20%, and an increase in the percentage of remote busy acquires from 6% to about 60%. Regardless of configuration, the percentage of remote idle acquires is constant at about 20% because of the large amount number of fine grained patch locks. As a result of these changes 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.

Locks in `raytrace` are accessed in a manner similar to the locks used in `radiosity`. In the 4- and 8-node configurations, acquires to the locks that protect a memory pool are 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 the number of nodes increases, both sets of locks become 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% better than the shared memory locks for the same reasons as in `radiosity`.

In `mp3d`, the degree of contention for locks is extremely low – 2% or fewer of the acquires are to busy locks. In addition, there is very little reuse, so most acquires 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 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 due to (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. 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. 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.

In `Spark98`, locks have good temporal locality due to the way that processes are assigned work. 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. As the number of nodes and/or network latency increases, however, MP-LOCKs perform up to 163% better than shared memory locks.

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 lock ownership can be forwarded in a single message. To achieve similar performance, shared memory locks would require special hardware shared memory protocols not present in modern machines. 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 18% 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.

5 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 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[13] and MCS locks[9], 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 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.

6 Acknowledgements

We would like to thank Leigh Stoller and Mark Swanson who provided invaluable assistance in using *Direct Deposit* protocol. Al Davis and Michael Scott provided helpful comments on this work. We also thank the members of *Avalanche* project at the University of Utah and the referees for many useful suggestions.

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 *Proc. of the 15th SOSP*, December 1995.

[2] G. Buzzard, D.Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the hamlyn sender-managed interface architecture. In *Proc. of the 2nd OSDI*, October 1996.

[3] J. Carter, A. Davis, R. Kuramkot, C.-C. Kuo, M. Swanson, and L. Stoller. Avalanche: A communication and memory architecture for scalable parallel computing. TR UUCS-95-022, Univ. of Utah, 1995.

[4] A. Kagi, D. Burger, and J. Goodman. Efficient synchronization: Let them eat qolb. In *Proc. of the 24th ISCA*, May 1997.

[5] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message-passing and shared-memory; early experience. In *Proc. of the 1993 PPoPP*, pp. 54–63, May 1993.

[6] C.-C. Kuo, J. Carter, and R. Kuramkote. MP-LOCKs: Replacing h/w synchronization primitives with message passing. TR UUCS-98-021, Univ. of Utah, November 1998.

[7] J. Kuskin and D. O. et al. The Stanford FLASH multiprocessor. In *Proc. of the 21st ISCA*, pp. 302–313, May 1994.

[8] B.-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proc. of the Sixth ASPLOS (ASPLOS-VI)*, pp. 25–35, October 1994.

[9] 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.

[10] D. O'Hallaron, J. Shewchuk, and T. Gross. Architectural implications of a family of irregular computations. In *Proc. of the Fifth HPCA*, pp. 80–89, Feb. 1998.

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

[12] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21st ISCA*, pp. 325–336, Apr. 1994.

[13] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proc. of the 11th ISCA*, pp. 340–347, May 1984.

[14] J. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. TR CSL-TR-91-469, Stanford Univ., Apr. 1991.

[15] L. Stoller and M. Swanson. Direct deposit: A basic user-level protocol for carpet clusters. TR UUCS-95-003, Univ. of Utah, March 1995.

[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd ISCA*, pp. 24–36, June 1995.