Reactive Synchronization Algorithms for Multiprocessors

Beng-Hong Lim and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Synchronization algorithms that are efficient across a wide range of applications and operating conditions are hard to design because their performance depends on unpredictable run-time factors. The designer of a synchronization algorithm has a choice of *protocols* to use for implementing the synchronization operation. For example, candidate protocols for locks include test-and-set protocols and queueing protocols. Frequently, the best choice of protocols depends on the level of contention: previous research has shown that test-and-set protocols for locks outperform queueing protocols at low contention, while the opposite is true at high contention.

This paper investigates reactive synchronization algorithms that dynamically choose protocols in response to the level of contention. We describe reactive algorithms for spin locks and fetch-and-op that choose among several shared-memory and message-passing protocols. Dynamically choosing protocols presents a challenge: a reactive algorithm needs to select and change protocols efficiently, and has to allow for the possibility that multiple processes may be executing different protocols at the same time. We describe the notion of *consensus objects* that the reactive algorithms use to preserve correctness in the face of dynamic protocol changes.

Experimental measurements demonstrate that reactive algorithms perform close to the *best* static choice of protocols at all levels of contention. Furthermore, with mixed levels of contention, reactive algorithms outperform passive algorithms with fixed protocols, provided that contention levels do not change too frequently. Measurements of several parallel applications show that reactive algorithms result in modest performance gains for spin locks and significant gains for fetch-and-op.

1 Introduction

Shared-memory multiprocessors usually provide read-modify-write hardware primitives for process synchronization, leaving the synthesis of higher-level synchronization operations to software synchronization algorithms. However, synchronization algorithms that are efficient across a wide range of applications are hard to design because their performance depends on run-time factors that are hard to predict, such as contention and waiting times.

In particular, a synchronization algorithm has a choice of *protocols* to implement the synchronization operation, and a choice of *waiting mechanisms* to wait for synchronization delays. For example, candidate protocols for locks include test-and-set and queueing protocols, and waiting mechanisms include spinning and blocking. Frequently, the best protocol depends on the level of contention, while the best waiting mechanism depends on waiting time.

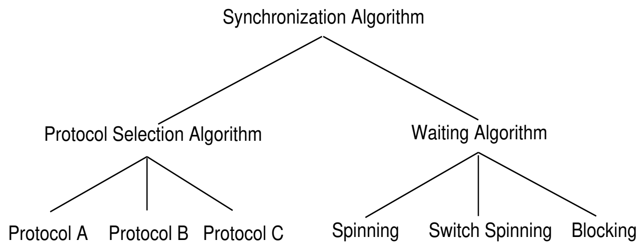

Figure 1: *The components of a reactive synchronization algorithm.*

For robust performance, a reactive synchronization algorithm dynamically selects protocols and waiting mechanisms in response to run-time factors. Figure 1 illustrates the components of a reactive synchronization algorithm. It is decomposed into a protocol selection algorithm and a waiting algorithm. The protocol selection algorithm is responsible for choosing a protocol to implement the synchronization operation, while the waiting algorithm is responsible for choosing waiting mechanisms to wait for synchronization latencies.

This paper focuses on reactive synchronization algorithms that choose protocols dynamically, but that assume spin waiting as a fixed waiting mechanism. Previous research [9, 14] has already demonstrated the utility of dynamically choosing waiting mechanisms. However, to the best of our knowledge, there has not been any experimental research on the feasibility and performance benefits of dynamically selecting synchronization protocols. We leave an investigation of simultaneously choosing protocols and waiting algorithms as future work.

As an example of the potential benefit of dynamically choosing protocols, consider the tradeoff between test-and-set and queueing protocols for spin locks. The test-and-set protocol acquires a lock with a *test-and-set* instruction and releases the lock with a *store* instruction. Although it is a simple and efficient protocol in the absence of contention, its performance degrades drastically under high contention. A remedy is a queueing protocol that constructs a software queue of waiters to reduce contention. However, queuing comes at the price of a higher latency in the absence of contention [2, 16]. The right choice between the two protocols depends on the level of contention experienced by the lock.

Figure 2 illustrates this tradeoff with typical overhead numbers for an acquire-release pair of a spin lock on the Alewife machine

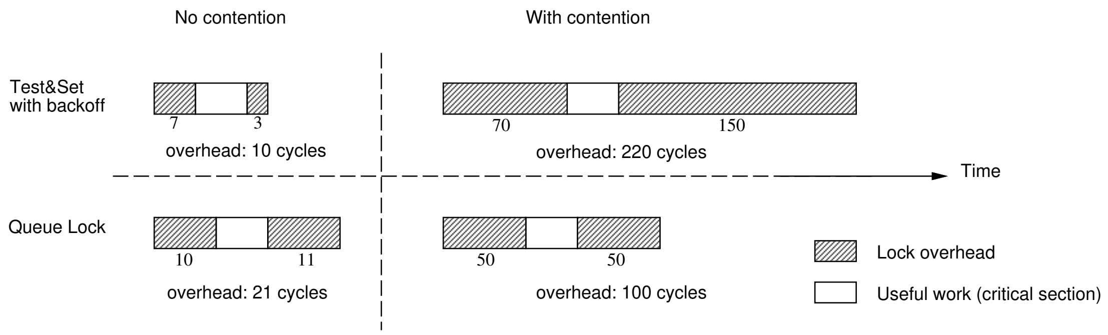

Figure 2: *A comparison of the overheads of an acquire-release pair of two different lock protocols at different levels of contention. Both protocols spin-wait. Without contention, the overhead is the latency of an acquire-release pair. With contention, the overhead is the time to pass ownership of the lock from a process to another.*

[1], a cache-coherent distributed-memory multiprocessor. Without contention, the shaded bars on each time-line represent the total time to execute a lock acquire and release. With contention, the shaded bars represent the time to pass ownership of the lock from the releasing processor to a waiting processor. Specifically, the left shaded bar is the time for some waiting processor to notice that the lock is free and acquire it. The right shaded bar is the time to release the lock. The test-and-set protocol has a lower overhead (10 vs. 21 cycles) when there is no contention, while the queue protocol has a lower overhead in the presence of contention (100 vs. 220 cycles).

This paper presents and analyzes reactive synchronization algorithms that choose among shared-memory and message-passing protocols in response to the level of contention. Dynamically selecting protocols presents a challenge: a reactive algorithm needs to choose and change protocols correctly and efficiently, while allowing for the possibility that multiple processes may be executing different protocols at the same time. In contrast, the choice of waiting mechanisms is a local operation that is independent of the state of other processes in the system. We introduce the notion of *consensus objects* that the reactive algorithms use to help ensure correct operation in the face of dynamic protocol changes.

Experimental measurements demonstrate that under fixed contention levels, reactive algorithms perform close to the *best* static choice of protocols at any level of contention. Furthermore, with mixed levels of contention, reactive algorithms outperform conventional *passive algorithms* with fixed protocols, provided that contention levels do not change too frequently. The reactive algorithms can also outperform passive algorithms in cases where there are a number of different synchronization objects, each with different levels of contention. Measurements of several parallel applications show that the reactive algorithms result in modest performance gains for spin locks and significant gains for fetch-and-op.

This research makes the following contributions:

1. To our knowledge, it is the first experimental investigation of the idea of dynamically choosing protocols to reduce synchronization overhead. Measurements from a multiprocessor simulator demonstrate its advantages.
2. It presents a method, based on the notion of *consensus objects*, for correctly and efficiently selecting synchronization protocols at run time. We use the method for dynamically selecting among several shared-memory and message-passing protocols for spin locks and fetch-and-op.
3. It has resulted in new algorithms for spin locks and fetch-and-op. The reactive spin lock has the scalable performance of queue locks while maintaining the low latency of test-and-set locks at low contention. The reactive fetch-and-op algorithm achieves the high throughput of a combining tree algorithm under high contention while maintaining the low latency of a non-combining algorithm at low contention.

The rest of the paper is organized as follows. Section 2 reviews conventional passive algorithms for spin locks and fetch-and-op. Section 3 presents reactive spin lock and fetch-and-op algorithms and introduces the notion of consensus objects to help change protocols efficiently. Section 4 presents experimental measurements and evaluates the performance of the reactive algorithms. Section 5 describes reactive algorithms that select between shared-memory and message-passing protocols. Section 6 considers the overhead of switching protocols. Section 7 concludes the paper.

2 Motivation

This section reviews conventional shared-memory spin-lock and fetch-and-op algorithms, and presents performance measurements of these passive algorithms to motivate the need for reactive algorithms. In Section 5 we will also consider message-passing protocols for spin locks and fetch-and-op.

2.1 Passive Spin-Lock Algorithms

Recent research has resulted in scalable spin-lock algorithms that alleviate the detrimental effects of memory contention [2, 8, 16]. The research demonstrated that test-and-set with randomized exponential backoff and queuing were the most promising spin-lock protocols.

Test-and-set with Exponential Backoff With a test-and-set protocol, a process requests a lock by repeatedly executing a *test-and-set* instruction on a boolean flag until it successfully changes the flag from false to true. In a variation, known as test-and-test-and-set [18], waiting processes read-poll the flag and attempt a *test-and-set* only when the flag appears to be false. A process releases a lock by setting the flag to false. To avoid excessive communication traffic under high contention, a waiting process introduces some delay in between accesses to the flag. In [2], Anderson demonstrated that randomized exponential backoff was an effective method for selecting the amount of delay. Henceforth, we will refer to test-and-set with exponential backoff simply as test-and-set locks, and test-and-test-and-set with exponential backoff simply as test-and-test-and-set locks.

Queue Locks Queue locks explicitly construct a queue of waiters in software. Memory contention is reduced because each waiter spins on a different memory location and only one waiter is signalled when the lock is released. Queue locks have the additional advantage of providing fair access to the lock. Several queue lock protocols were developed independently by Anderson [2], Graunke and Thakkar [8], and Mellor-Crummey and Scott [16]. In this paper, we use the MCS (Mellor-Crummey and Scott) queue lock because it has the best performance among the queue locks on our system.

The MCS queue lock maintains a pointer to the tail of a software queue of lock waiters. The lock is free if it points to an empty queue, and is busy otherwise. The process at the head of the queue owns the lock, and each process on the queue has a pointer to its successor. To acquire a lock, a process appends itself to the tail of the queue. If the queue was empty, the process owns the lock; otherwise it waits for a signal from its predecessor. To release a lock, a process checks to see if it has a waiting successor. If so, it signals that successor, otherwise it empties the queue. See [16] for further details.

2.2 Passive Fetch-and-Op Algorithms

Fetch-and-op is a powerful primitive that can be used to implement higher-level synchronization operations. When the operation is *combinable* [10], *e.g.*, fetch-and-add, combining techniques can be used to compute the operation in parallel. Several software algorithms can be used to implement fetch-and-op. We consider the following in this paper.

Lock-Based Fetch-and-Op A straightforward implementation of fetch-and-op is to protect the fetch-and-op variable with a mutual exclusion lock. In particular, either the test-and-set lock or the queue lock described above can be used here. To execute a fetch-and-op, a process acquires the lock, updates the value of the fetch-and-op variable, and releases the lock.

Software Combining Tree A drawback of lock-based implementations of fetch-and-op is that they may serialize fetch-and-op operations unnecessarily. Software combining techniques [21] can be used to compute the fetch-and-op in parallel. The idea is to combine multiple operations from different processes into a single operation whenever possible. The fetch-and-op variable is stored in the root of a software combining tree, and combining takes place at the internal nodes of the tree. If two processes arrive simultaneously at a node, their operations are combined. One of the processes proceeds up the tree with the combined operation while the other waits at that node. When a process reaches the root of the tree, it updates the value of the fetch-and-op variable and proceeds down the tree, distributing results to processes that it combined with on the way up. In this paper, we use the software combining tree algorithm for fetch-and-op presented by Goodman, Vernon and Woest in [5].

2.3 The Problem with Passive Algorithms

The problem with passive algorithms is that they fix their choice of protocols, and are thus optimized for a certain level of contention/concurrency at the synchronization operation. We measured

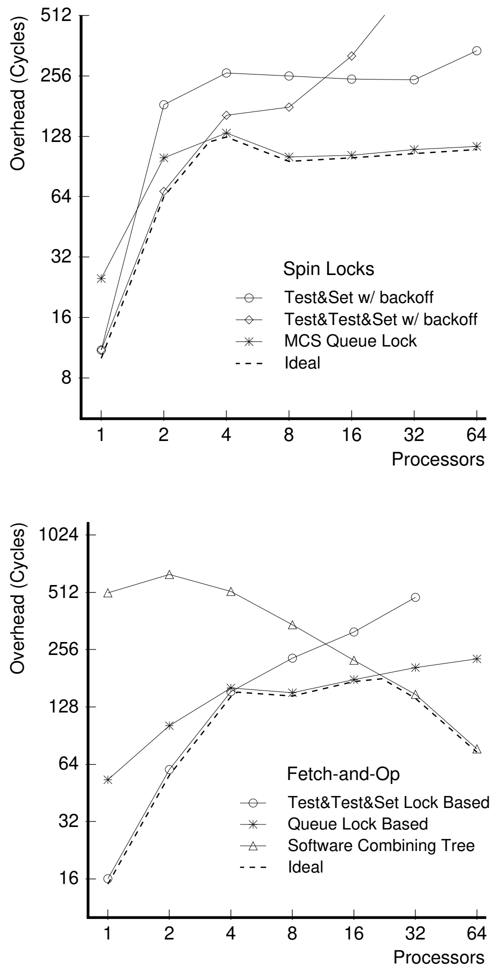

Figure 3: *Baseline performance of passive spin lock and fetch-and-op algorithms.*

the performance of various spin lock and fetch-and-op algorithms on a simulation of the Alewife multiprocessor. Figure 3 compares the performance of the best algorithms we found. For spin locks, each data point represents the average overhead incurred for each critical section with P processors contending for the lock. For fetch-and-op, each data point represents the average overhead incurred for each operation with P processors attempting to perform the operation. Section 4 provides more details on the experiments.

The spin lock results show that the MCS queue lock provides the best performance at high contention levels. However, it is twice as expensive as the test-and-set lock when there is no contention. The test-and-test-and-set lock has the lowest overhead at low contention levels, but does not scale well. The reason test-and-test-and-set does not scale as well as test-and-set is because on machines like Alewife without hardware-supported broadcast, cache updates or invalidations occur sequentially. Test-and-test-and-set has the effect of delaying a lock release as read-cached copies of the lock are serially updated or invalidated. These results indicate that we should use the test-and-test-and-set lock when contention is low and the MCS queue lock when contention is high.

The fetch-and-op results show that software combining parallelizes the overhead of fetch-and-op: as contention (and hence parallelism) increases, the average overhead drops. In contrast, the overhead increases with contention for the lock-based algorithms. However, when contention is low, software combining incurs a large overhead from having to traverse the combining tree, and the lock-based algorithms perform better. Here again, we have a contention-dependent choice of algorithms for fetch-and-op.

We would like performance to follow the ideal curves. As we will see in Section 4, our reactive algorithms perform very close to this ideal.

3 Reactive Algorithms

The previous section demonstrated the need for algorithms that automatically select protocols according to the level of contention experienced by a synchronization operation. Such algorithms will relieve the programmer from predicting run-time conditions to reduce synchronization overheads. In this section, we first present a reactive spin lock algorithm that selects between a test-and-test-and-set protocol and the MCS queue lock protocol as an example reactive algorithm. We then describe an efficient method for dynamically selecting protocols based on the notion of *consensus objects*, and apply the method towards the design of a reactive fetch-and-op algorithm.

3.1 Issues in Dynamically Selecting Protocols

In designing reactive algorithms that dynamically select among multiple synchronization protocols, we have to address three issues:

1. How does the algorithm efficiently detect which protocol to use? Since each synchronization operation must detect which protocol to use, this should not incur a significant cost.
2. How does the algorithm change protocols correctly and efficiently? A protocol change involves correctly updating the state of the reactive algorithm's protocols and notifying any participating processes of the change. It also needs to be efficient in cases where frequent protocol changes are expected.
3. When should the algorithm change protocols? A reactive algorithm needs to detect that the current protocol is unsuitable and decide if it is profitable to switch to another protocol.

The first two issues are concerned with providing efficient mechanisms for dynamic protocol selection, while the third is a policy issue.

The main difficulty of providing efficient mechanisms for dynamic protocol selection is that multiple processes may be trying to execute the synchronization operation at the same time, and keeping them in constant agreement on the protocol to use may be prohibitively expensive. The solution is to allow multiple processes to execute different protocols, but ensure that only processes that execute the correct protocol are allowed to succeed. Processes that execute the wrong protocol will retry the operation. This optimizes for the expected common case when the currently selected protocol is the right protocol. It assumes that contention levels do not oscillate between extremes so as to require frequent protocol changes. This assumption holds in the applications that we have looked at so far. Section 4 will analyze these applications.

The third issue of when to switch protocols depends on how contention levels vary over time, and on the architectural parameters

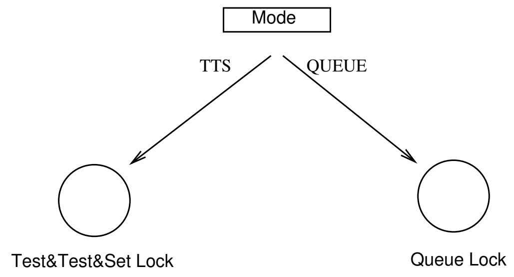

Figure 4: *Components of the reactive spin lock: a test-and-test-and-set lock, a queue lock, and a mode variable. The mode variable provides a hint to lock requesters on which sub-lock to use.*

of the multiprocessor. Since the performance crossover points for different synchronization protocols are architecture dependent, the switching policy will have to be tuned for each machine architecture. However, we emphasize that this tuning is application independent.

3.2 A Reactive Spin Lock

Our reactive spin lock combines the low latency of a test-and-test-and-set lock with the scalability and fairness properties of the MCS queue lock. It does so by dynamically selecting between the test-and-test-and-set protocol and the MCS queue lock protocol. Figure 4 illustrates the components of the reactive lock. It is composed of two sub-locks (a test-and-test-and-set lock and an MCS queue lock), and a mode variable.

Intuitively, the reactive spin-lock algorithm works as follows. Initially, the test-and-test-and-set lock is free, the queue lock is busy, and the mode variable is set to `TTS`. A process acquires the reactive lock by acquiring either the test-and-test-and-set lock or the queue lock. The algorithm ensures that the test-and-test-and-set lock and queue lock are never free at the same time, so that at most one process can successfully acquire a sub-lock. A process releasing the reactive lock can choose to free either the test-and-test-and-set lock or the queue lock, independent of which sub-lock it acquired.

Detecting which protocol to use. The mode variable provides a hint on which sub-lock to acquire. On a cache-coherent multiprocessor, assuming infrequent mode changes, the mode variable will usually be read-cached so that checking it incurs very little overhead. If the mode variable is `TTS`, a process attempts to acquire the test-and-test-and-set lock. Otherwise, if the mode variable is `QUEUE`, a process attempts to acquire the queue lock.

The mode variable is only a hint because there exists a race condition whereby the correct protocol to use can change in between when a process reads the mode variable to when that process executes the protocol indicated by the mode variable. However, since the reactive algorithm ensures that the test-and-test-and-set lock and queue lock are never free at the same time, processes that execute the wrong protocol will simply find the corresponding sub-lock to be busy. Such processes will either re-check the mode variable, or receive a retry signal while waiting, and retry the synchronization operation with a different protocol.

Correctly and efficiently changing protocols. The reactive algorithm needs to update the state of the mode variable and the sub-locks consistently when changing protocols. We restrict protocol changes to lock holders to ensure atomicity. When changing from TTS mode to QUEUE mode, the reactive lock holder changes the value of the mode variable to QUEUE, then releases the queue lock, leaving the test-and-test-and-set lock in a busy state. When changing from QUEUE mode to TTS mode, the lock holder changes the value of the mode variable to TTS, signals any waiters on the queue to retry, then releases the test-and-test-and-set lock, leaving the queue lock in a busy state.

When to change protocols. The reactive algorithm estimates the level of lock contention to decide if the current lock protocol is unsuitable. The reactive spin lock changes from the test-and-test-and-set protocol to the queuing protocol when the number of failed test-and-set attempts experienced by a process while acquiring the lock exceeds a threshold. In the opposite direction, it changes from the queueing protocol to the test-and-test-and-set protocol if a process finds the queue to be empty for some number of consecutive lock acquisitions. This provides some hysteresis to lessen the likelihood of oscillating and thrashing between protocols. While this simple method worked well for the experiments in this paper, more sophisticated methods may be desired. We describe several possible alternatives at the end of this section.

3.3 Consensus Objects

We now describe a general mechanism for dynamic protocol selection. Our method for efficiently selecting and changing synchronization protocols requires that *each* protocol being selected has the following properties:

1. Each protocol has a unique object, called a *consensus object*, that has to be accessed atomically in order to complete the protocol. This consensus object may be marked as valid or invalid.

2. A process must be able to complete execution of the protocol after accessing the consensus object, independently of other processes that subsequently access the consensus object.

3. The state of a protocol that represents the state of the synchronization operation is modified only by a process that has atomic access to the consensus object.

The first property eases the task of ensuring that only one protocol is valid at any time. The second property allows a process that executes an invalid protocol to abort and preserve the consistency of the invalid protocol's state. The third property eases the task of updating the state of the newly valid protocol during protocol changes.

The protocols selected by the reactive spin lock trivially satisfy these properties. The consensus objects are the locks themselves. For the test-and-test-and-set lock protocol, the consensus object is simply the test-and-test-and-set lock itself, and for the queue lock protocol, the consensus object is simply the queue lock itself.

If a reactive algorithm is selecting among protocols that satisfy the properties listed above, then processes do not have to be kept in constant agreement on the protocol in use. We only need to ensure that the only valid consensus object is the one associated with the protocol currently in use. This allows a process to safely execute an invalid protocol because that process will ultimately access the protocol's invalid consensus object. At that point, that process will complete execution of the invalid protocol and retry the synchronization operation with another protocol. Thus, the consensus object allows a process to decide whether it is executing the right protocol.

In order to switch to a new protocol, the mode variable and the state of the new protocol that represents the state of the synchronization operation has to be atomically updated. We allow protocol changes to be performed only by a process that has acquired atomic access to the consensus object of the currently valid protocol. This ensures that at most one process will be changing the mode variable and updating the state of the new protocol at any time. To change from protocol A to protocol B, a process acquires atomic access to protocol B's invalid consensus object, invalidates protocol A's consensus object and releases it, updates the state of protocol B to reflect the current state of the synchronization operation, and changes the mode variable to point to protocol B. Finally, it validates protocol B's consensus object and releases it.

3.4 A Reactive Fetch-and-Op Algorithm

Using the concept of consensus objects described above, we designed a reactive algorithm for fetch-and-op that chooses among the following three protocols:

1. A variable protected by a test-and-test-and-set lock.

2. A variable protected by a queue lock.

3. A software combining tree by Goodman *et al.* [5].

Each of these protocols satisfies the properties listed in Section 3.3, allowing the reactive fetch-and-op algorithm to use mechanisms similar to the reactive spin lock algorithm for dynamically selecting and changing protocols. For the first two protocols, the consensus objects are the locks protecting the centralized variables. For the combining tree, the consensus object is the lock protecting the root of the combining tree.

Detecting which protocol to use. As in the reactive spin lock algorithm, a mode variable ushers processes to the fetch-and-op protocol currently in use. The reactive algorithm ensures that at most one of the fetch-and-op protocols' consensus objects is valid at any time. A process that accesses an invalid lock while executing one of the centralized protocols simply retries the fetch-and-op with another protocol.

For the combining tree, a process that accesses an invalid root has a set of waiting processes that it combined with on the way to the root. Thus, that process has to complete the protocol by proceeding down the combining tree and notifying those waiting processes to retry the fetch-and-op operation. These processes will in turn notify processes that they combined with to retry the operation.

Correctly and efficiently changing protocols. Only a process with exclusive access to the currently valid consensus object is allowed to change protocols. Recall that when changing to another protocol, the state of the target protocol needs to be updated to represent the current state of the synchronization operation. For fetch-and-op, the state of the synchronization operation is the current value of the fetch-and-op variable. In each of the three protocols, this state is represented by a variable that is modified only by processes with exclusive access to a protocol's consensus object. Thus the state of the target protocol can be easily updated by updating the value of this variable.

When to change protocols. The reactive fetch-and-op algorithm changes from the test-and-test-and-set lock protocol to the queue lock protocol after some number of failed test-and-set attempts. It changes from the queue lock protocol to the combining tree protocol when the waiting time on the queue exceeds a time limit for a number of successive fetch-and-op requests. Since the queue is FIFO, the waiting time on the queue provides a good estimate of the level of contention. In the other direction, the reactive algorithm changes from the combining tree protocol to the queue lock protocol when a number of successive fetch-and-op requests reach the root without combining with a sufficient number of other fetch-and-op requests. It changes from the queue lock protocol to the test-and-test-and-set protocol if the queue lock's queue is empty for a number of successive fetch-and-op requests. Again, these simple criteria worked well for the experiments described in the next section, but more sophisticated policies may be desirable. We describe several alternative policies here.

3.5 Policies for Switching Protocols

A reactive algorithm that finds itself using a sub-optimal protocol needs to decide whether to switch to a better protocol. Because switching protocols incurs a significant fixed cost, this decision depends on the future behavior of contention levels. The algorithms described above used hysteresis to reduce the probability of thrashing among protocols. Although a thorough analysis of switching policies is beyond the scope of this paper, we briefly mention a few alternative policies here.

Since the decision to switch is an instance of an on-line problem where decisions have to be made without knowledge of future contention levels, a possible policy is to use competitive techniques [15] to decide when to switch protocols. With competitive techniques, the worst-case performance of the switching policy can be bounded by a constant.

Another possible policy is to use aging to compute a weighted average of the history of contention levels, and select the protocol that is optimal for the average contention level. Also, the reactive algorithm can detect if mode changes are required too frequently. If so, the algorithm can give up on being reactive and use the best protocol for the average contention level experienced thus far.

4 Experimental Results

In this section, we present experimental measurements that compare the performance of the reactive spin lock and fetch-and-op algorithms with conventional passive algorithms on the MIT Alewife multiprocessor [1]. The Alewife architecture is representative of a large-scale multiprocessor with cache-coherent distributed shared memory. Nodes in the multiprocessor communicate via messages through a 2-D mesh network. Memory is physically distributed among the nodes, and cache coherence is maintained using the LimitLESS cache coherence protocol [4]. Cache-coherent atomic *fetch-and-store* is directly supported in hardware.

At the time this research was performed, the Alewife multiprocessor was still being implemented. We relied on an accurate cycle-by-cycle simulation of the machine to gather the data presented in this section. The simulations assume a processor 33MHz clock frequency. Recently, a 16-node Alewife machine became operational, and we will present data from experiments run on the real hardware in Section 6.

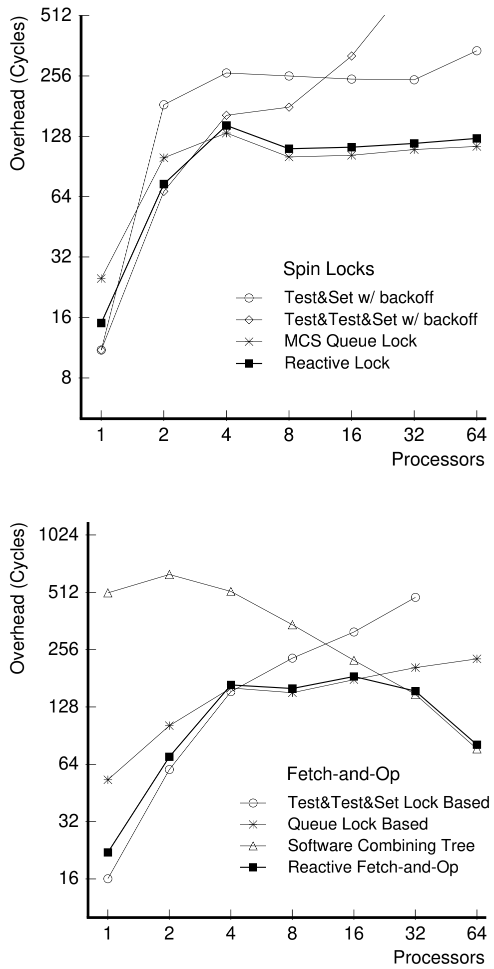

Figure 5: *Baseline performance of spin lock and fetch-and-op algorithms.*

4.1 Baseline Performance

In this experiment, we measured the overhead incurred per synchronization operation at different levels of contention. This is the measure used in previous research on synchronization algorithms. The results show that the reactive algorithms succeed in selecting the right protocol to use, and are close to the performance of the best passive algorithms at all levels of contention. Figure 5 compares the baseline performance of the algorithms.

Spin Locks Each processor executes a loop that acquires the lock, executes a 100-cycle critical section, releases the lock, and delays for a random period between 0 and 500 cycles. The 100-cycle critical section models a reasonably small critical section when contention is involved: protected data has to migrate between caches and it takes about 50 cycles to service a remote cache miss. The delay between lock acquisitions forces the lock to migrate between caches when there is contention. This test program is similar to that used by Anderson [2].

Each data point represents the average lock overhead per critical section with P processors contending for the lock. The overhead is measured by subtracting the amount of time the test would have

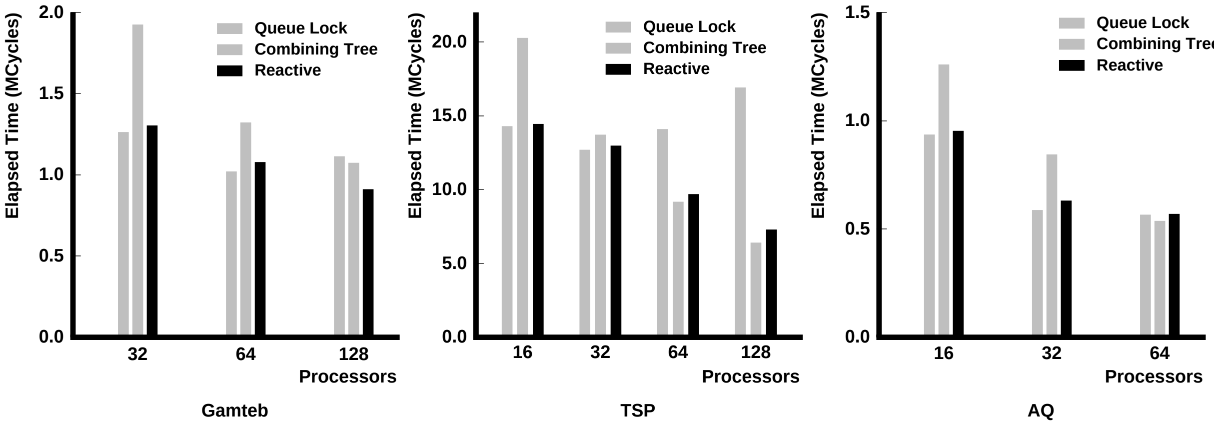

Figure 6: *Execution times for applications using different fetch-and-op algorithms.*

taken, given zero-overhead spin locks, from the actual running time. Without contention, the overhead represents the latency of an acquire-release pair. With contention, the overhead represents the time to pass ownership of the lock from one process to another. The reactive algorithm selects the test-and-test-and-set protocol at the one- and two-processor data points, and selects the queueing protocol at all other data points.

Fetch-and-Op Each processor executes a loop that executes a fetch-and-increment, then delays for a random period between 0 and 500 cycles. The delay between increment requests forces the fetch-and-increment variable to migrate between caches when there is contention. We used a radix-2 combining tree with 64 leaves.

Each data point represents the average overhead incurred per fetch-and-increment operation with P processors contending for the operation. The overhead is measured by subtracting the amount of time the test would have taken, given zero-overhead fetch-and-increment operations, from the actual running time. The reactive algorithm selected the test-and-test-and-set lock-based protocol for the one- and two-processor data points, the queue-based protocol for the 4-16 processor data points, and the combining tree protocol for the 32- and 64-processor data points. The reactive fetch-and-op algorithm combines the advantages of each of its component protocols: it has low latency when contention is low and high throughput when contention is high.

4.2 Application Performance

The baseline performance numbers provide a precise measure of the overheads of the synchronization algorithms. We now investigate the impact of the synchronization algorithms on several parallel applications that use spin locks and fetch-and-op. The applications are written in C and parallelized with library calls. For each application, we vary the synchronization algorithm used and measure the execution time on various numbers of processors.

Figure 6 presents the execution times for applications that use fetch-and-op. We exclude the execution times for the test-and-test-and-set lock-based fetch-and-op protocol: they were either slightly better or much worse than the execution times for the queue-based protocol. The combining trees are radix-2 and have as many leaves as the number of processors in each experiment.

Overall, the results show that the choice of fetch-and-op algorithms has a significant impact on the execution times, and that the reactive fetch-and-op algorithm selects the right protocol to execute in all cases. They demonstrate the utility of having a reactive algorithm select the protocol to use. To better understand the results, we describe the characteristics of each application.

Gamteb Gamteb [3] is a photon transport simulation based on the Monte Carlo method. In this simulation, Gamteb was run with an input parameter of 2048 particles. Gamteb updates a set of nine interaction counters using fetch-and-increment.

On 32 and 64 processors, contention at all nine interaction counters were such that the queue-based protocol for fetch-and-op exhibits the best performance. The reactive algorithm selected the queue-based protocol for all the counters. On 128 processors, contention at one of the counters was high enough to warrant a combining tree. The reactive algorithm selected the combining tree protocol for that counter and the queue-based protocol for the other eight counters. This allowed the reactive algorithm to outperform the passive algorithms that use the same protocol for all of the counters.

Traveling Salesman Problem (TSP) TSP solves the traveling salesman problem with a branch-and-bound algorithm. Processes extract partially explored tours from a global task queue and expand them, possibly generating more partial tours and inserting them into the queue. In this simulation, TSP solves an 11-city tour. To ensure a deterministic amount of work, we seed the best path value with the optimal path. The global task queue is based on an algorithm for a concurrent queue in [7] that allows multiple processes simultaneous access to the queue. Fetch-and-increment operations synchronize access to the queue.

Contention for the fetch-and-increment operation in this application depends on the number of processors. With 16 and 32 processors, the queue-based fetch-and-op protocol is superior to the combining tree, but the opposite is true with 64 and 128 processors. The reactive algorithm selected the queue-based protocol at 16 and 32 processors, and the combining tree protocol at 64 and 128 processors.

Adaptive Quadrature (AQ) AQ performs numerical integration of a function with the adaptive quadrature algorithm. It proceeds by continually subdividing the range to be integrated into smaller ranges. A free processor dequeues a range to be integrated

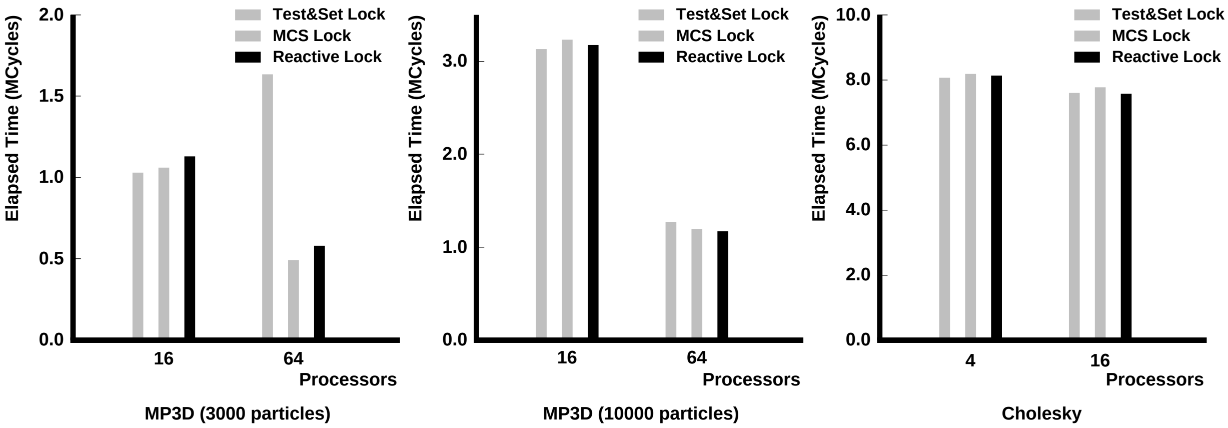

Figure 7: *Execution times for applications using different spin lock algorithms.*

from a global task queue. Depending on the behavior of the function in that range, the processor may subdivide the range into two halves, evaluate one half and insert the other into the queue. Here, AQ is used to integrate the function $(sin(\pi i))^{30}$ in the range $(0, 30)$.

The task queue implementation is the same as the one used in TSP. However, the computation grain size of each task in the parallel queue are larger compared to TSP, resulting in lower contention for the fetch-and-increment operation. At 16 and 32 processors, the queue-based fetch-and-op protocol is superior to the combining tree, but at 64 processors, both the queue-based and combining-tree protocols perform about equally. The reactive algorithm selected the queue-based protocol at 16 and 32 processors, and the combining tree protocol at 64 processors.

Figure 7 presents the execution times for applications with spin locks. Overall, the results show that while high contention levels may be a problem for the test-and-set lock, the higher latency of the MCS queue lock at low contention levels is not a significant factor. Computation grain sizes between critical sections for these applications are large enough to render the higher latency of the queue lock insignificant. Thus, the reactive spin lock yielded limited performance benefits over the MCS queue lock.

Nevertheless, the reactive spin lock still achieves performance that is close to the best passive algorithm. It should be useful for applications that perform locking frequently and at a very fine grain such that lock latencies becomes significant.

MP3D MP3D is part of the SPLASH parallel benchmark suite [19]. For this simulation, we use problem sizes of 3,000 and 10,000 particles with the locking option turned on. We measured the time taken for 5 iterations. Locks are used in MP3D for atomic updating for cell parameters, where a cell represents a discretization of space. Contention at these locks is typically low. A lock is also used for atomic updating of collision counts at the end of each iteration. Depending on load balancing, contention at this lock can be high.

The higher latency of the MCS queue lock under low contention was not significant. On the other hand, the poor scalability of the test-and-set lock significantly increased execution time for 3,000 particles on 64 processors. The reactive lock selected the test-and-test-and-set lock protocol for atomic updating of cell parameters, and selected the queue lock for updating collision counts.

Cholesky Cholesky is also part of the SPLASH parallel benchmark suite. It performs Cholesky factorization of sparse positive definite matrices. Speed and space limitations of the Alewife simulator limited us to factorizing small matrices with limited amounts of parallelism. Therefore we do not intend this simulation to be indicative of the speedup characteristics of the SPLASH benchmark. In this simulation, we factorize an 866x866 matrix with 3189 non-zero elements. As in MP3D, we see that the higher latency of the MCS lock has a negligible impact on execution times.

5 Reactive Algorithms and Message-Passing Protocols

We now consider reactive algorithms that select between shared-memory and message-passing protocols. Recent architectures for scalable shared-memory multiprocessors [11, 12, 17] implement the shared-memory abstraction on top of a collection of processing nodes that communicate via messages through an interconnection network. They allow software to bypass the shared-memory abstraction and access the message layer directly.

These architectures provide an opportunity to use message-passing protocols for synchronization operations. The advantage of using message-passing to implement synchronization operations over shared-memory is that under high contention, message-passing results in more efficient communication patterns, and atomicity is easily provided by making message handlers atomic with respect to other message handlers [20].

For example, fetch-and-op can be implemented by allocating the fetch-and-op variable in a memory location. To perform a fetch-and-op, a process sends a message to the home processor associated with that memory location. The message handler computes the operation on that location and returns the result with its reply. This results in the theoretical minimum of two messages to perform a fetch-and-op: a request and a reply. Contrast this with shared-memory protocols for fetch-and-op that require multiple messages to ensure atomic updating of the fetch-and-op variable.

While message-passing protocols can outperform corresponding shared-memory protocols under high contention, the fixed overheads of message sends and receives make message-passing protocols more expensive than corresponding shared-memory protocols when contention is low. Once again, we have a contention-dependent choice to make between protocols.

Using the method based on consensus objects, we designed re-

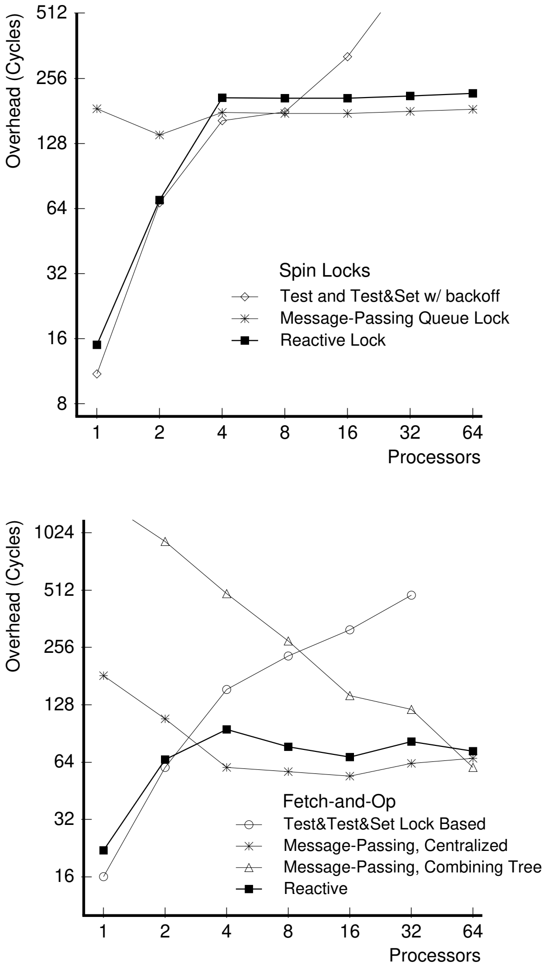

Figure 8: *Baseline performance comparing shared-memory and message-passing protocols for spin locks and fetch-and-op. The reactive algorithms select between the shared-memory and message-passing protocols.*

active algorithms for spin locks and fetch-and-op that select between shared-memory and message-passing protocols. The reactive spin-lock algorithm selects between a test-and-test-and-set lock protocol and a message-passing queue lock protocol. The message-passing queue lock is implemented by designating a processor as a lock manager. To request a lock, a process sends a message to the lock manager and waits for a reply granting it the lock. The lock manager maintains a queue of lock requesters and responds to lock request and release messages in a FIFO manner.

The reactive fetch-and-op algorithm selects between a test-and-test-and-set lock based protocol, a centralized message-passing fetch-and-op protocol (described in the example above), and a message-passing combining-tree protocol. The message-passing combining tree protocol uses messages to traverse the combining tree. To execute a fetch-and-op, a process sends a message to a leaf of the tree. After polling the network to detect messages that it may combine with, a message handler relays a message to its parent. In this way, messages combine and propagate up to the root of the combining tree where the operation is performed on the fetch-and-op variable.

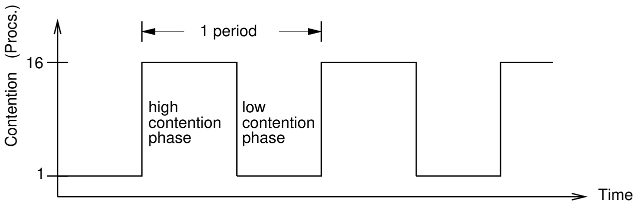

Figure 9: *The dynamic test periodically varies the level of contention to force the reactive lock to undergo mode changes.*

Figure 8 presents the baseline performance of these algorithms. Like the reactive algorithms that select between purely shared-memory protocols, these reactive algorithms also succeed in selecting the right protocol for a given level of contention. As a demonstration of the advantage of using message-passing over shared-memory protocols, note that under high contention, the message-passing fetch-and-op protocols result in lower overheads and correspondingly higher throughputs than the shared-memory fetch-and-op protocols. Although the numbers show that the message-passing queue lock is always inferior to the shared-memory MCS queue lock on Alewife, the reverse may be true on architectures with different communication overheads and levels of support for shared-memory.

6 Overhead of Switching Protocols

In designing the reactive algorithms, we assumed that contention levels do not vary between extremes so as to cause frequent protocol changes. We now investigate the behavior of the reactive algorithms when this assumption does not hold. To expose the overhead of changing protocols frequently, we run a test program that periodically switches between phases of no contention and high contention. A 16-node Alewife machine recently became operational, allowing us to perform this test on the real hardware.

Figure 9 illustrates how the level of contention varies during the test. In the low-contention phase, a single processor executes a loop that acquires the lock, executes a 10-cycle critical section, releases the lock, and delays for 20 cycles. In the high contention phase, 16 processors concurrently execute a loop that acquires the lock, executes a 100-cycle critical section, releases the lock, and delays for 250 cycles. We measure the time for the test program to execute a fixed number of critical sections.

Figure 10 presents the results of this experiment. In each graph, we plot the execution time of the test program, normalized to the MCS queue lock. We vary the test program along two dimensions: i) the number of locks acquired in each period (locks per period), and ii) the percentage of locks that are acquired under high contention (% contention). The number of locks acquired per period controls the length of a period, and affects how frequently the reactive lock has to switch protocols. Each period causes to reactive lock to switch protocols twice.

If contention levels do not vary too frequently (towards the right end of each graph), the results show that the test-and-set lock outperforms the MCS queue lock when contention is rare (10% contention). When contention dominates (90% contention), the MCS queue lock outperforms the test-and-set lock. In both cases, the reactive lock approaches the performance of the better of the two passive algorithms. When there is some mix of low and high contention (30% contention), neither the test-and-set lock nor the MCS

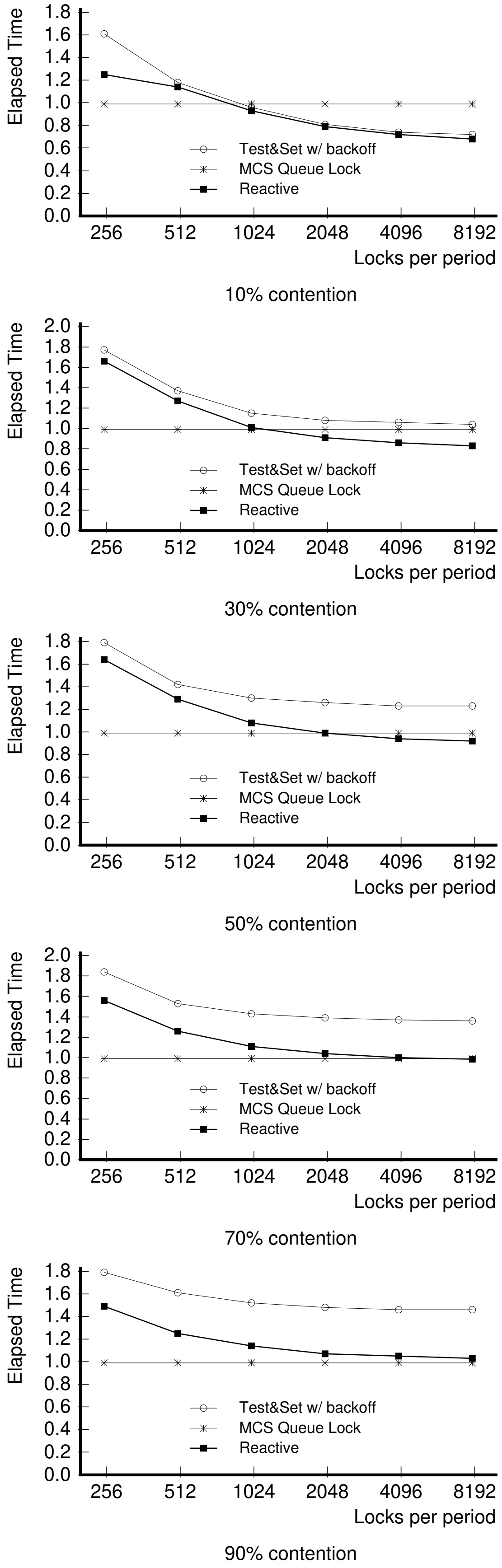

Figure 10: *Elapsed times for the dynamic test, normalized to the MCS Queue Lock.*

queue lock has a clear advantage over the other. By continuously selecting the better protocol, the reactive lock outperforms both the test-and-set and MCS queue locks.

If contention levels vary too frequently (towards the left end of each graph), the overhead of switching protocols dominates and the performance of the reactive lock suffers. In the experiments, the performance of the reactive spin lock begins to deteriorate when forced to change protocols as frequently as every 1000 critical sections. However, the reactive lock is still always better than the worst static choice of protocols even under such extreme circumstances. It is interesting to note that the performance of the test-and-set protocol deteriorates when contention levels change frequently. This happens because the test-and-set protocol does not handle bursty arrivals of lock requesters as well as the MCS queue lock protocol.

While we do not expect contention levels to exhibit such pathological behavior in practice, more intelligent switching policies, such as those described in Section 3.5, may mitigate the detrimental effect of frequently changing contention levels.

7 Summary and Conclusions

The performance of synchronization algorithms depends on unpredictable run-time factors, such as contention. Recent research designed synchronization algorithms that perform well under high contention, but at the price of higher overhead under low contention. We would like to use the best algorithm for a given level of contention. However, the level of contention is hard to predict. As a solution, this paper proposes reactive synchronization algorithms that dynamically make the best choice of protocols.

This paper demonstrates the feasibility and performance benefits of selecting synchronization protocols in response to the level of contention. While the idea of dynamically selecting protocols is intuitively appealing, it was not clear prior to this research if the complexity of managing multiple protocols would be prohibitively expensive. We described reactive algorithms for spin locks and fetch-and-op, and described a method based on consensus objects for efficiently selecting protocols in those algorithms. We also demonstrated that reactive algorithms can be used to select between shared-memory and message-passing protocols.

Experiments show that the performance of the reactive algorithms is close to the best of any of the passive algorithms at all levels of contention. Furthermore, when contention levels are mixed, the reactive algorithm outperforms the passive algorithms, as long as contention levels do not vary too frequently. Reactive algorithms also outperform passive algorithms when there are a number of synchronization objects, each with different levels of contention. Measurements of several applications show that the reactive algorithms result in modest performance gains for spin locks and significant gains for fetch-and-op.

Several multiprocessor architectures, *e.g.*, the Wisconsin Multicube [5] and Stanford DASH [13], include hardware support for queue locks. Hardware queuing has the advantage of low lock latency in the absence of contention. However, application measurements indicate that the additional latency of the MCS queue lock is not a significant factor unless locking is performed frequently at a very fine granularity. If latency is a concern, the reactive spin lock algorithm will provide the low latency of a test-and-set lock with the scalability of a queue lock. This reduces the motivation for providing hardware support for queue locks.

The NYU Ultracomputer [6] includes hardware support for combining fetch-and-add operations through the interconnection network. Although software combining algorithms have been proposed as an alternative to hardware combining, they have the disadvantage of high latencies. By maintaining low latency when contention is low and high throughput when contention is high, reactive fetch-and-op algorithms provide a more viable alternative.

Interested readers can obtain a pseudocode listing of the reactive spin lock algorithm via anonymous ftp from hing.lcs.mit.edu (directory pub/bhlim/reactive).

Acknowledgments

We would like to thank David Chaiken, Stuart Fiske, Maurice Herlihy, Wilson Hsieh, Kirk Johnson, David Kranz, John Kubiatowicz, Deborah Wallach, and Donald Yeung for helpful comments on this work. The referees provided many insightful comments as well. This research was supported in part by by ARPA grant #N00014-91-J-1698, NSF grant # MIP-9012773, and a NSF Presidential Young Investigator Award.

References

[1] Anant Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper appears as MIT/LCS Memo TM-454, 1991.

[2] Thomas E. Anderson. The Performance Implications of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] P.J. Burns *et al.* Vectorization of Monte-Carlo Particle Transport: An Architectural Study using the LANL Benchmark "Gamteb". In *Proc. Supercomputing '89*, New York, NY, November 1989. IEEE/ACM.

[4] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[5] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 64–75, April 1989.

[6] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing a MIMD Shared-Memory Parallel Machine. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[7] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[8] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, pages 60–70, June 1990.

[9] Anna Karlin, Kai Li, Mark Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. In *13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 41–55, October 1991.

[10] Clyde Kruskal, Larry Rudolph, and Marc Snir. Efficient Synchronization on Multiprocessors with Shared Memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, October 1988.

[11] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *International Supercomputing Conference (ICS) 1993*, Tokyo, Japan, July 1993. IEEE.

[12] Jeffrey Kuskin *et al.* The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994. IEEE.

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

[14] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[15] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive Algorithms for On-line Problems. In *Proceedings of the 20th Annual Symposium on Theory of Computing*, pages 322–333, Chicago, IL, May 1988. ACM.

[16] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[17] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994. IEEE.

[18] Z. Segall and L. Rudolph. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedions of the 11th Annual International Symposium on Computer Architecture*, pages 340–347. IEEE, June 1984.

[19] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[20] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[21] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.