

A Performance Evaluation of Lock-Free Synchronization Protocols

Anthony LaMarca*

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

In this paper, we investigate the practical performance of lock-free techniques that provide synchronization on shared-memory multiprocessors. Our goal is to provide a technique to allow designers of new protocols to quickly determine an algorithm's performance characteristics. We develop a simple analytical performance model based on the architectural observations that memory accesses are expensive, synchronization instructions are more expensive, and that optimistic synchronization policies result in wasted communication bandwidth which can slow the system as a whole. Using our model, we evaluate the performance of five existing lock-free synchronization protocols. We validate our analysis by comparing our results with simulations of a parallel machine. Given this analysis, we identify those protocols which show promise of good performance in practice. In addition, we note that no existing protocols provide insensitivity to common delays while still offering performance equivalent to locks. Accordingly, we introduce a protocol, based on a combination of existing lock-free techniques, which satisfies these criteria.

1 Introduction

An important and relatively new class of concurrent objects are those which synchronize without locks. The largest barrier preventing the adoption

*The author was supported by an AT&T Fellowship. This paper appears in the 1994 Principles of Distributed Computing (PODC)

lock-free synchronization protocols is their performance in practice. In the common case, lock-free protocols have higher latency and can generate more contention than traditional locks. While there have been efforts to build more efficient protocols and to optimize existing ones, little has been done to describe their performance in general. In this paper, we present a model with which we investigate the practical performance of several previously published lock-free synchronization protocols. Our performance model reflects three important architectural characteristics. First, that remote memory accesses are more expensive than local, cached memory accesses. Second, that synchronization instructions can be more expensive than normal memory references. Third, that optimistic synchronization policies result in unsuccessful thread updates which consume communication bandwidth and slow the progress of the successful threads.

Given our analysis, we note that no existing protocols provide insensitivity to common delays while still offering performance equivalent to locks. Accordingly, we introduce a protocol, based on a combination of existing lock-free techniques, which provides low latency and insensitivity to preemption delays.

1.1 Background

Threads running on shared-memory multiprocessors coordinate with each other via shared data structures called *concurrent objects*. In order to prevent the corruption of these concurrent objects, threads need a mechanism for synchronizing their access. Typically, this synchronization is provided with locks protecting critical sections, ensuring that at most one thread can access an object at a time. Concurrent

threads are difficult to reason about, and critical sections greatly simplify the building of a correct shared object. While much work has been done to improve the performance of locks [4, 13, 19], critical sections are not well suited for asynchronous systems. The delay or failure of threads holding locks can severely degrade performance and cause problems such as *convoying*, in which parallelism is not exploited as threads move together from lock to lock, *priority inversion*, in which a high priority thread cannot make progress because it needs a lock being held by a low priority thread that has been preempted, and *dead-lock*, which occurs when all active threads in the system are waiting on locks [24]. In the extreme case, the delay of a single thread holding a single lock can prevent all other threads from making progress. Sources of these delays include cache misses, remote memory accesses, page faults and scheduling pre-emptions.

Recently, researchers have proposed concurrent objects which synchronize without the use of locks. As these objects are built without locks, they are free from the aforementioned problems. In addition, lock-free objects can offer progress guarantees. A lock-free object is *non-blocking* if it guarantees that some thread completes an operation in a finite number of steps. A lock-free object is *wait-free* if it guarantees that each thread completes an operation in a finite number of steps [14]. Intuitively, the non-blocking property says that adversarial scheduling cannot prevent all of the threads from making progress. The wait-free property is stronger and says that starvation is not possible regardless of the potential interleavings of operations.

There has been a considerable amount of research supporting the practical use of lock-free synchronization. Herlihy and Moss have proposed architectural changes which enable efficient lock-free computing [16]. Bershad has described a technique which uses operating system support to simulate important synchronization instructions when they are unavailable in the architecture [6]. Numerous lock-free implementations of specific data-structures have been proposed including Wing and Gong's object library [22, 23], Massalin's lock free objects [18] and Anderson and Woll's union-find object [2]. In addition, a number of software protocols have been developed which generate lock-free concurrent objects given a traditional sequential implementation of the

object. Herlihy's small object protocol and wait-free protocol [15], Alemany and Felten's protocols [1], and Barnes' caching method [5] fall in this category. Since these protocols build concurrent objects with no knowledge of the object's semantics, we call them *black-box* synchronization protocols. These black-box protocols are of practical interest since they can be applied to any sequential object and could be automatically applied by a compiler. Unfortunately, these protocols do not perform as well as traditional lock-based synchronization in practice. For the remainder of the paper, we will be focusing on the performance characteristics of black-box synchronization protocols.

1.2 Organization

The remainder of the paper is organized as follows. In section 2 we discuss three important architectural characteristics and their effect on protocol performance. In order to demonstrate their importance, we combine these three characteristics into a simple analytical model which we describe in section 3. In section 4 we briefly describe five existing lock-free protocols and evaluate their performance in our model. In addition, we introduce a new synchronization protocol which offers insensitivity to thread pre-emptions without sacrificing performance. In section 6, we validate our model by comparing its results to those produced by parallel-machine simulations. In section 7, we present our conclusions.

2 Performance Issues

A common criticism of lock-free synchronization techniques is that while they offer real-time and fault-tolerant benefits over locks, these benefits are not realized when there are no thread delays and that, in actual use, they perform poorly. In general, lock-free protocols have higher latency and generate more memory contention than locks. In this section, we describe three important architectural characteristics which affect the performance of lock-free synchronization protocols.

1. *Remote memory accesses are more expensive than local, cached memory accesses*

With few exceptions, modern machines have fast local caches and accessing these caches is one

to two orders of magnitude faster than accessing main memory. In addition, depending on the coherency protocol, local references do not put a load on the communication medium used to support shared memory, unlike remote references. Considering the trend of processor speed increasing relative to memory speed, this factor will be increasingly important. Given that caching can have such a large impact on performance, we would not expect a PRAM-style analysis in which all memory accesses have unit cost to accurately predict performance.

The idea of treating remote and local references differently is not a new one. Many models, including Snyder's CTA and more recently the logP model include this distinction [11, 20]. The notion of reducing non-local memory references during synchronization has also been previously investigated. Anderson and Mellor-Crummey and Scott consider local versus remote accesses when explaining the contention caused by *Test-and-Test&Set* locks and in the design of their queue-based spin locks [4, 19].

2. Synchronization instructions are more expensive than normal memory references

Lock-free synchronization protocols make use of synchronization instructions such as *Test&Set*, *Compare&Swap* and the combination of *Load Linked* and *Store Conditional*. On current machines these synchronization instructions incur a cycle cost much higher than that of a normal memory reference. For example, on a Dec 3000-400 with a 130 Mhz Alpha 21064 CPU [12], we observed a cost of 140 cycles for the pair of *Load Linked* and *Store Conditional* instructions, 3.5 times more expensive than a normal uncached read. This property is not exclusive to this particular architecture; synchronization instructions for many modern processors incur similar costs [6, 7]. We do not distinguish synchronization from non-synchronization instructions because of an inherent difference in complexity, but because of the implementation differences that occur in practice. This distinction is also important when the necessary synchronization instructions are unavailable on an architecture and must be simulated by the operating system [6]. In these situations, the code executed to simulate the desired instruction can

take much longer than a single memory operation and needs to be taken into account. Lastly, distinguishing synchronization instructions from non-synchronization instructions is important on distributed shared memory systems such as Munin [10] or Midway [8] or shared memory multiprocessors such as Dash [17] which support a consistency model looser than sequential consistency. In these systems, synchronization instructions invoke the coherency mechanism which in turn results in costly communication.

Again, we find that analyzing synchronization algorithms in a PRAM-style model which assigns all instructions unit cost can introduce unnecessary inaccuracy.

3. Optimistic synchronization policies result in unsuccessful thread updates which consume communication bandwidth and slow the progress of the successful threads

In order to be non-blocking, a number of the lock-free protocols behave optimistically, that is, all threads proceed as if they will succeed. Once threads realize that their update has failed, they either begin another attempt or they cooperate in order to help the successful thread [5, 15]. This optimistic policy results in the waste of machine resources when there is contention for a shared object. Processor cycles are wasted that could possibly be used by another thread. More importantly, communication bandwidth is wasted which in turn slows down the successful thread. As more unsuccessful threads contend for the communication bandwidth, the progress of the system as a whole can be crippled. This is what Alemany and Felten refer to as "useless parallelism" [1]. Herlihy attempts to alleviate this problem by using exponential backoff to reduce the number of unsuccessful updates [14].

The degradation caused by an optimistic policy is important and should be quantified in a good performance model.

The performance impact of these characteristics motivates us to develop an analytical model which takes them into account.

3 Performance Model

In this section we present a simple performance model that reflects the memory costs, synchronization costs and wasted work costs discussed in section 2. It is intended that our model be simple enough to evaluate algorithms quickly, yet still provide good insight into practical performance. This model can be used to explain the performance of existing protocols, determine how changes in architecture will affect protocol performance, and serve as a guide for designers of new lock-free protocol.

3.1 Model Variables

Our performance model measures the amount of work done by a particular protocol in order to complete a single update to a shared object. In our model, we assume that the data caches start out cold and that all instruction fetches hit in the cache. We divide instructions into three categories: local cached instructions such as loads which hit in the cache, instructions that access memory such as loads which miss in the cache, and synchronization instructions such as *Compare&Swap* and *Store Conditional*. We assign local cached instructions cost 0 in order to both keep the model simple and to reflect their low cost. Non-synchronization instructions which access memory have normalized cost of 1. Lastly, in order to reflect their higher cost, we assign synchronization instructions cost C .

In our model, N denotes the number of threads in the system. S represents the size, in words, of the sequential object's state. R and W denote the number of non-overlapping reads and writes respectively which are made to the object in the course of an operation. Since we only consider non-overlapping reads and writes, $R + W$ is less than or equal to S , but could be significantly smaller. Enqueuing to a 20 word stack, for instance, might have $S = 20$, $R = 1$ and $W = 2$.

3.2 Workloads

In our model, we consider the performance protocols in the presence of three different adversaries, each of which reflects a different workload. The first case we consider is when there is no contention and a single thread applies an operation to the concurrent object.

Torrellas *et al.* found that on a 4 CPU multiprocessor running System V, threads accessing the six most frequently accessed operating system data structures found them unlocked from 85 to 99 percent of the time [21]. While this says nothing about user code or large multiprocessors, it does suggest that well written systems are designed to minimize the contention on shared objects. In order to measure the latency of a protocol in this case, we introduce a weak adversary which we call *best* that allows a single thread to execute its operation to completion without blocking.

While we do not expect high contention to be the common case, it is still important to know how a protocol's performance degrades when an object is accessed by multiple threads concurrently. In order to model high contention, we include a *bad* scenario in which all N threads are trying to apply an operation concurrently and the adversarial scheduler can arbitrarily interleave their instructions.

A number of lock-free protocols rely on the operating system to execute code on a thread's behalf when it block, either on I/O or due to a scheduler pre-emption. For these protocols, there is an important distinction between the scheduler blocking a thread and the scheduler simply not allowing a thread to run. In order to measure the effect that an extremely powerful adversary can have on these protocols, we also include a *worst* scenario. In our *worst* scenario, all N threads are active and the adversarial scheduler can arbitrarily interleave their instructions and cause them to block. Once a thread is blocked, the scheduler has no obligation to wake the thread up.

4 Applying the Model

We now briefly describe five existing lock-free protocols which we will evaluate in our model. The earliest technique for building concurrent objects is Herlihy's *small object protocol* [14, 15]. Herlihy's small object protocol can be extended to build wait-free objects by applying a technique called *operation combining*. We call the resulting protocol *Herlihy's wait free protocol*. Both of Herlihy's protocols have the drawback of useless parallelism and high copying overhead. Alemany and Felten proposed solutions to both problems which rely on support from the operating system [1]. In order to reduce useless parallelism, Alemany and Felten developed a proto-

| Method | Best Case | Bad Case | Worst Case |
|-----------------------------------|---------------------------------|------------------------------------|---------------------------------------|
| Herlihy's small object | $S + C + 3$ | $N(S + C + 3)$ | $N(S + C + 3)$ |
| Herlihy's wait-free | $S + 2N + C + 3$ | $N(S + 2N + C + 3)$ | $N(S + 2N + C + 3)$ |
| Barnes' Caching Method | $R(4C + 3) + W(6C + 4) - C - 1$ | $N(R(4C + 3) + W(6C + 4) - C - 1)$ | $N(R(4C + 3) + W(6C + 4) - C - 1)$ |
| Aleman and Felten's solo protocol | $S + 3C + 4$ | $N(C + 1) + S + 2C + 3$ | $N(S + 4) + C(2N + \frac{N(N-1)}{2})$ |
| A and F's solo w/ logging | $R + 2W + C + 1$ | $R + 2W + N(C + 1)$ | ∞ |
| Spin-lock | $R + W + C + 1$ | $R + W + N(C + 1)$ | ∞ |

Table 1: Total amount of work done to complete a single operation.

col which explicitly limits the number of concurrent threads to a small number K . When $K = 1$, they call this their *solo protocol*. In order to reduce copying overhead, Alemany and Felten added logging and rollback to their solo protocol, and they call this their *solo protocol with logging*. The final lock-free protocol we will evaluate is *Barnes' caching method*. Of all of the black-box lock-free techniques, only Barnes' caching method produces implementations which allow threads to make progress in parallel.

Table 1 shows the total amount of work done for a single update operation by the black-box techniques evaluated using our model. We show expressions for Herlihy's small object and wait-free protocol, Barnes' caching method, Alemany and Felten's solo protocol and solo protocol with logging. For comparison, we also include expressions for the amount of work done by a *test-and-Compare&Swap* spin-lock. We now briefly describe the derivation of several expressions in Table 1.

Herlihy's small object protocol gets its best case performance because the shared pointer must first be *load-linked* (at cost 1)¹, the object needs to be copied (at cost S), two reads are done to validate the copy (at cost 2), and after the local computation, a *store-conditional* of cost C is incurred. In the worst case, the scheduler can force all N threads to *load-linked* the pointer, copy and validate the object, apply their operation and attempt the *store-conditional* for a total cost of $N(S + C + 3)$ per successful operation.

Herlihy's wait-free protocol differs from the small object protocol in that, for all scenarios, the threads

¹We treat *load-linked* as a normal memory operation in our calculations.

incur overhead due to the copying of the result table and the scanning of the announcement table[15].

In Barnes' and Alemany and Felten's work, only pseudocode of the protocols are provided. In these cases, we consider an efficient implementation with straightforward optimizations.

In the *best* scenario, Alemany and Felten's solo protocol is worse than Herlihy's small object protocol because it incurs additional work $(C + 1)$ to check and increment the counter and work C to decrement it. In the *worst* case, the scheduler can cause the thread counter to be a point of contention by repeatedly allowing a new thread in, blocking it and forcing all the waiting threads to attempt to increment the counter. This results in an additional $C \frac{N(N-1)}{2}$ work.

In Barnes protocol, threads first perform their operations on a local cached copy at cost $(R + W)$. In order to cooperate safely, Barnes' protocol effectively creates a program which the threads interpret together. The program is first installed at a cost of C . The program is then interpreted at a cost of $(4C + 2)$ per operation read and $(6C + 3)$ per operation write. An optimization can be made for the first read or write resulting in a reduction of $(2C + 1)$. This totals $R(4C + 3) + W(6C + 4) - C - 1$. In the *worst* and *bad* scenarios, all N threads can be made to perform all $R(4C + 3) + W(6C + 4) - C - 1$ work. The rest of the expressions were derived using similar reasoning.

4.1 The solo-cooperative protocol

Choosing among the existing synchronization protocols involves a tradeoff between theoretical

```

Concurrent_Apply(op_name, op_args) {
  repeat {
    repeat {
      old_op_ptr := op_ptr;
    }until (old_op_ptr <> 0);
  }until (Compare&Swap(op_ptr, old_op_ptr, 0) = Success);

  if (old_op_ptr <> 1) {
    Complete_Partial_Operation(old_op_ptr);
  }
  ret_val := Sequential_Apply(op_name, op_args);
  op_ptr := 1;
  return(ret_val);
}

```

Figure 1: Pseudocode for main body of the solo-cooperative protocol

progress guarantees and practical performance. At one extreme, Herlihy's wait-free protocol offers the strongest progress guarantees and incurs substantial overhead in doing so. Herlihy's small object protocol gives up the wait-free property for the non-blocking property and a decrease in latency. Similarly, Barnes' caching method gives up the wait-free property in exchange for non-blocking and parallelism which could increase performance. Alemany and Felten's solo protocol performs better under contention but is not robust to processor failure. Their solo protocol with logging offers even lower latency but gives up non-blocking and is simply tolerant of common thread delays. Lastly, there are finely tuned locks which offer good performance but no tolerance of delays.

Given our performance concerns, we are interested in protocols which offer the same performance as locks and offer tolerance to some delays. Because most existing protocols have evolved from the more theoretical protocols, it is not surprising to find that such a protocol does not currently exist. We now describe a protocol that offers good performance in practice and is insensitive to preemption delays. In order to achieve this, we combine the idea of Barnes-style thread cooperation with the single active thread used in Alemany and Felten's solo protocol.

The main problem with thread cooperation is that inefficiencies are introduced when protecting the threads from each other. By having only one thread active at a time, however, cooperation can be used

with no overhead in the common case of an operation that does not block. In our *solo-cooperative* protocol, a single thread at a time is allowed to update the shared object, similar to locking. This single thread updates the object without making a copy and without logging the changes being made. In order to provide insensitivity to delays, we include a mechanism which allows waiting threads to help finish the work of a blocked thread. If a thread blocks during an operation, its state is stored in the object and its ownership of the object is released. A new thread can then use the stored state to finish the partially completed operation and begin to apply its own changes. Anderson uses a technique similar to this in the protection of his user-level scheduler in his scheduler activations work [3].

Like Alemany and Felten's protocols, we rely on support from the operating system to execute code on behalf of a thread when it blocks and unblocks. This support requires changes to the operating system or the use of a system like Anderson's scheduler activations.

In our solo-cooperative protocol, the object is protected by a variable called *op_ptr* which has is functionally similar to a lock. If *op_ptr* is 0, the object is "locked"; if *op_ptr* is 1 the object is free; and for all other values, the object is free, but there is a partially finished operation to be finished and *op_ptr* points its description. To update the shared object, a thread first reads *op_ptr* and *Compare&Swaps* 0 for a non-zero value, guaranteeing exclusive access. If

| Method | Best Case | Bad Case | Worst Case |
|------------------|-----------------|--------------------|------------|
| Solo-cooperative | $R + W + C + 1$ | $R + W + N(C + 1)$ | ∞ |
| Spin-lock | $R + W + C + 1$ | $R + W + N(C + 1)$ | ∞ |

Table 2: Total amount of work done to complete a single operation.

the thread sees that the old value of op_ptr is 1, the thread applies its operation, sets op_ptr back to 1 and returns. If the old value of op_ptr is not 1, the thread completes the partially finished operation and subsequently applies its own operation. Pseudocode for the body of the protocol appears in Figure 1.

When a thread blocks during an operation, the system executes code on the thread’s behalf which bundles up the operations state (stored in the thread’s stack and registers) and stores a pointer to this in op_ptr . This has the effect of releasing op_ptr making additional updates to the shared object possible even though the executing thread has blocked.

When a thread that blocked during an operation is unblocking, it checks its control structure to see if its operation has already been completed by another thread. If the waking thread’s operation has not been completed, it will re-acquire op_ptr and will finish its operation. If the waking thread’s operation has been completed, it will read the operation’s result from its control structure and will continue execution after the operation.

In the case that a new thread acquires op_ptr and finds a partially complete operation, it cooperates by loading the blocked thread’s registers and continues the execution of the partial operation. On completion of the blocked thread’s operation, the cooperating thread writes the operation’s result in the blocked thread’s control structure and returns to apply its own operation.

This solo-cooperative protocol has the problem that if a thread blocks on I/O, such as a page fault, all of the threads in the system that attempt to cooperate will also block on this fault. While this may seem like a major flaw, it may have little impact in practice. In general we expect contention for an object to be low and that there will be no waiting threads. In the case that there are waiting threads, there is a good chance that the waiting threads will also need data that is on the faulting page. While it is possible that additional progress could be made which would

go unexploited by this protocol, we expect that this would not happen sufficiently often to be a problem.

An evaluation of our protocol in our model is shown in Table 2. For comparison the expression for the spin-lock is repeated from Table 1. Note that our solo-cooperative protocol executes no more memory operations or synchronization instructions than the spin-lock when threads do not block.

4.2 Evaluation

From Table 1, we see that Herlihy’s protocols and Alemany and Felten’s solo protocol incur overhead S due to their need to copy the entire object. The overhead renders these protocols impractical for objects with a large amount of state, such as a thread stack. However, for objects with a small amount of state, such as a shared counter, these protocols can be competitive in practice.

In the table, we also see that Barnes caching method relies heavily on synchronization instructions. Barnes claims that although a large amount of overhead is incurred per operation instruction, critical sections are designed to be short and simple. This technique could conceivably perform better than a copying protocol if the objects were large and the operations consisted of only a few instructions.

In the high-contention *bad* case, we see that large amounts of memory contention can be generated by Barnes’ and Herlihy’s protocols. In order to make these protocols viable in the presence of a large number of active threads, they need a concurrency restriction mechanism like exponential backoff. In the scenarios in which threads do not block, we see that Alemany and Felten’s solo protocol, our solo-cooperative protocol and the spin-lock provide the combination of low latency and low contention. Interestingly, these are the same three protocols which are neither non-blocking nor wait-free. As can be seen in the *worst* case, the strong adversary can keep all three of these protocols from making progress.

In the case of the spin-lock, the scheduler simply needs to allow a thread to acquire the lock and then blocks the thread. If the scheduler never wakes this thread up, no other threads can make progress and the protocol is deadlocked. While this can easily be caused by an adversarial scheduler, it can also happen in a multiprocessor due to process or processor failure.

While immune to deadlock, Alemany and Felten’s solo protocol with logging can be made to livelock by the adversarial scheduler. In the solo protocol with logging, when a thread blocks, its partial operation is first undone by the operating system using the log, the thread is then removed from the critical section and another thread is allowed to begin its operation. By repeatedly allowing threads to make progress and then blocking them before completion the scheduler can cause the protocol to process indefinitely without completing an operation. Thus, while retaining many of the benefits of the lock-free protocols, Alemany and Felten’s solo protocol with logging is neither wait-free nor non-blocking. This livelock behavior is not exclusive to an adversarial scheduler. If a thread’s operation is too long to execute in a single scheduling quantum, the thread will always block in the operation, and this can cause livelock.

Our solo-cooperative protocol can also be forced to livelock by the strong adversary. Thus our protocol is also neither wait-free nor non-blocking. Recall that in our protocol, the state of a blocked thread’s partially completed operation is stored under *op_ptr*. Upon encountering a partially complete operation, a thread will try to load the state of this operation and finish it. To force livelock, the scheduler first allows a thread to partially complete an operation and then forces it to block. The scheduler can then repeatedly allow a new thread to begin loading the state of the partially completed operation, and then block it before it makes any progress on the operation, thus causing livelock. This behavior seems limited, however, to our strong adversary and we have no reason to expect this to occur in practice.

5 Validation

Our performance model was designed to make the evaluation of protocol performance easy and fairly accurate. In order to verify its accuracy, we now

compare the results produced by our model with results produced by a parallel machine simulator. In our simulations we compare Herlihy’s small object protocol and wait-free protocol, Alemany and Felten’s solo protocol and solo protocol with logging, our solo-cooperative protocol, and for comparison, a *test-and-Compare&Swap* spin-lock. Although queue-based spin locks have some performance advantages over spin-locks, we chose a *test-and-Compare&Swap* spin-lock because of its simplicity and low latency. We did not simulate Barnes’ caching protocol due to its implementation complexity. In order to obtain information about execution performance, we ran the protocols in Proteus, a parallel machine simulator developed at MIT [9]. Proteus is an execution driven simulator which takes as input augmented C programs and outputs results from a parallel run of the program.

In our simulations, Proteus was configured to model a bus-based multiprocessor running the Goodman cache-coherency protocol. In order to reflect relative instruction costs in our simulations, memory instructions were chosen to be 6 times as expensive as local cached instructions and synchronization instructions were 12 times as expensive ($C = 2$). During our simulation, each thread ran on its own processor and thread delays did not occur, thus the special-case code for the protocols using operating system support was never executed. Our simulations consisted of a number of threads, varying from 1 to 20, alternately updating a shared object and performing 200 instructions worth of local work. We simulated the protocols using both a shared counter and a circular queue as the shared object.

In our model and our simulations we do not include exponential backoff in any of the protocols. Exponential backoff reduces resource contention by reducing concurrency, but does not change the relationship between concurrency and contention in a given protocol. Given that we are focusing on the interaction between concurrency and resource contention, there was no additional insight to be gained by including exponential backoff.

In our first set of simulations, threads alternately update a shared counter and perform local work until a total of 4096 updates have been done. Figure 2 shows the number of simulated cycles per update, varying the number of threads from 1 to 20. The graph shows that initially, as the parallelism in

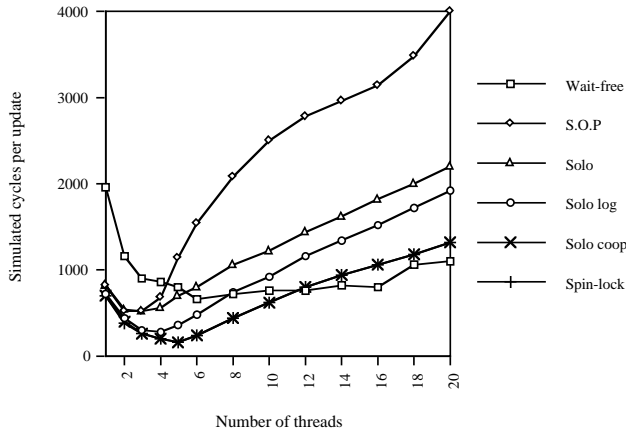


Figure 2: Simulated shared counter

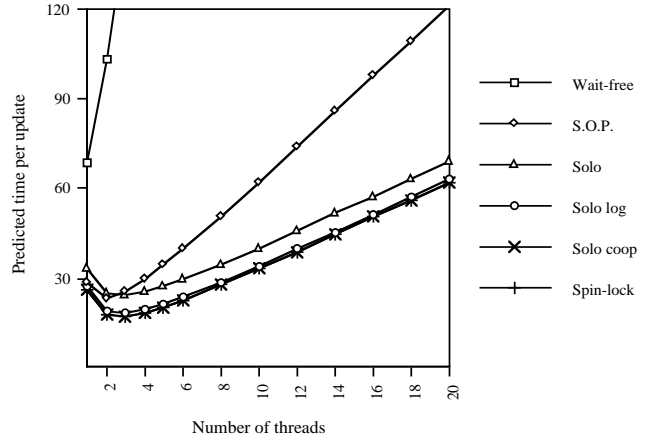


Figure 3: Predicted shared counter

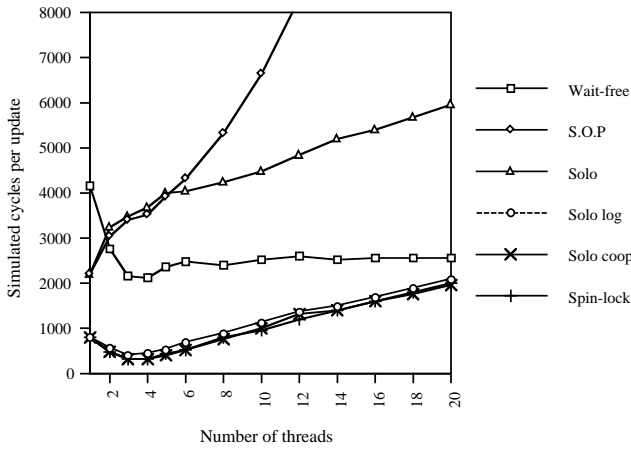


Figure 4: Simulated circular queue

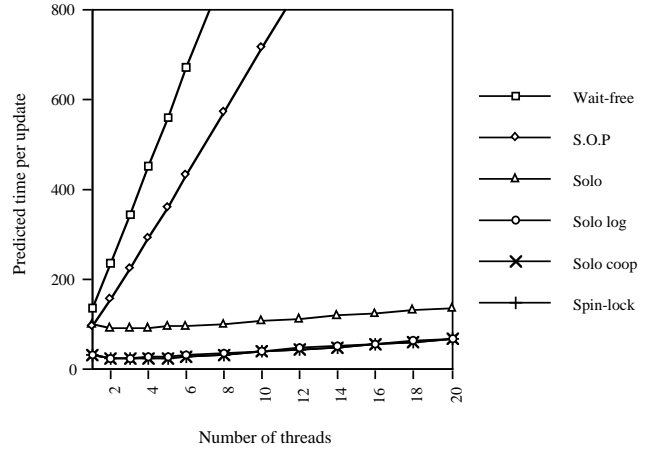


Figure 5: Predicted circular queue

the system is exploited, additional processors improve performance. As the number of processors further increases, however, performance degrades as memory contention increases due to synchronization overhead.

For our shared counter, the model variables are $S = 1$, $R = 0$, and $W = 1$. Recall that our model predicts the amount of work that a protocol can generate for a single successful update. In order to compare our model to the simulation results, we need to convert this work quantity to a time quantity. In addition we need to factor in the local work performed by the threads. For our bus-based machine, we make the simple assumption that the protocol overhead is forced to serialize and that the local work can proceed in parallel. Let the amount of work predicted by

our bad case adversary in Table 1 be $W(N)$. Given a total of M updates and local work L per update, we expect $MW(N) + ML$ total work. Dividing protocol overhead by M and local work by MN yields a predicted $W(N) + \frac{L}{N}$ time per update. A graph of this expression for the simulated protocols is given in Figure 3.

In our second set of simulations, threads alternately enqueue or dequeue from a 64 entry circular queue and perform local work. As before, the threads perform a total of 4096 updates per run and we vary the number of threads from 1 to 20. Figure 4 shows the results for the simulations. Our circular queue has model variables $S = 66$ and $R = 1$, $W = 3$ for enqueue and $R = 2$, $W = 2$ for dequeue. Since roughly the same number of enqueues

and dequeues occur, we model the structure as having $S = 66$, $R = 1.5$ and $W = 2.5$. We again make the assumption that local work is performed in parallel and that protocol overhead serializes. Figure 5 shows the results predicted by our model.

Despite the simplicity of our model, the predictions are fairly accurate. With the exception of Herlihy's wait-free protocol, the ordering of relative performance is the same in both cases. In both the simulations and our model the spin-lock and our solo-cooperative protocol performed well, providing low latency and degrading reasonably under high load. The predicted performance for the solo protocol with logging differs from the simulation results for the shared counter, but remain ordinally correct and the shape of the curves are similar. Both the simulations and the model predictions show that Herlihy's small object protocol without backoff generates significant contention and that the single active thread in Alemany and Felten's solo protocol does reduce this contention.

The performance of Herlihy's wait-free protocol is seriously mispredicted by our model. While our model predicts the protocol to perform poorly, the simulator shows that it scales well as concurrency increases. This inaccuracy stems from the basic assumption our model makes that for each update no threads are initially accessing the object at which point all N threads arrive to perform an update. While this assumption works well for the other protocols, it inaccurately models the behavior of the wait-free protocol. In Herlihy's wait-free protocol, threads register their intended operations in the object before beginning their update. In this way, given the protocol's cooperation policy, even if a thread's update attempts are unsuccessful, its operation must complete within 2 successful updates [15]. Under high contention, Herlihy's wait-free protocol amortizes the cost of the object copy and table scan over a number of successful updates, resulting in its good scaling. This amortization is not taken into account by our bad case scenario which assumes that $N - 1$ of the N threads fail every update. This oversight could be remedied by introducing an adversary which forces the maximum work per update under a steady-state high contention scenario in which a number of threads continually occupies the object.

With this exception, our model accurately reveals the important performance characteristics of the pro-

ocols we examined. The model should serve as a useful tool for those designing and analyzing lock-free synchronization algorithms.

6 Conclusions

There are a number of lock-free synchronization protocols which offer guarantees about progress and which eliminate the problems traditionally associated with locks. These protocols can be applied to any sequential object and could be applied by compilers. Unfortunately, these benefits can come at the cost of decreased performance. We have quantified the performance of these protocols by building and validating a model in which to evaluate them. While our model is fairly simple, it reflects three important architectural characteristics. First, that remote memory accesses are more expensive than local, cached memory accesses; second, that synchronization instructions can be more expensive than normal memory references; and third, that optimistic synchronization policies result in unsuccessful thread updates which consume communication bandwidth and slow the progress of the successful threads. Using our model, we evaluated five lock-free protocols and illustrated important performance characteristics of each. None of these protocols provide delay insensitivity while retaining the low latency of locking. Accordingly, we presented a lock-free protocol that combines Alemany and Felten's single thread with Barnes-style cooperation to provide low latency and insensitivity to preemption delays.

Of the protocols evaluated, those using operating system support have the most promise in practice. Alemany and Felten's solo protocol with logging and our solo-cooperative protocol offer the combination of low latency in the common case and low memory contention in the high load case. This performance comes with the disadvantage that neither of these protocols are non-blocking or wait-free. These protocols do, however, retain many of the lock-free benefits, and in the absence of failures, should perform well.

References

- [1] J. Alemany and E. W. Felten. Performance Issues in Non-blocking Synchronization on Shared-Memory Multiprocessors. In *11th Annual ACM Sym-*

- posium on Principles of Distributed Computing*, August 1992.
- [2] R. J. Anderson and H. Woll. Wait-Free Parallel Algorithms for the Union-Find Problem. In *23rd Annual ACM Symposium on Theory of Computing*, pages 360–370, 1991.
- [3] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 95–107, October 1991.
- [4] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
- [5] G. Barnes. A method for implementing lock-free data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms & Architecture*, June 1993.
- [6] B. Bershad. Practical Considerations for Non-Blocking Concurrent Objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, May 1993.
- [7] B. Bershad, D. Redell, and J. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, October 1992.
- [8] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON '93)*, pages 528–537, February 1993.
- [9] E. A. Brewer, A. Colbrook, C. N. Dellarocas, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. In *1992 ACM Sigmetrics*, pages 247–8, June 1992.
- [10] J. B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 152–164, October 1991.
- [11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.
- [12] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor, Hardware Reference Manual*, 1992. Order Number: EC-N0079-72.
- [13] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.
- [14] M. Herlihy. A Methodology for Implementing Highly Concurrent Structures. *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, March 1990.
- [15] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects., 1991. CRL Technical Report 91/10.
- [16] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *20th Annual International Symposium on Computer Architecture*, 21(2):289–300, May 1993.
- [17] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *19th Annual International Symposium on Computer Architecture*, pages 92–105, May 1992.
- [18] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 191–200, December 1989.
- [19] J. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1), February 1991.
- [20] L. Snyder. Type Architecture, Shared Memory and the Corollary of Most Potential. *Annual Review of Computer Science*, 1:289–318, 1986.
- [21] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, October 1992.
- [22] J. M. Wing and C. Gong. A Library of Concurrent Objects and Their Proofs of Correctness., 1990. Technical Report CMU-CS-90-151, Carnegie Mellon University.
- [23] J. M. Wing and C. Gong. Testing and Verifying Concurrent Objects. *Journal of Parallel and Distributed Computing*, 17(2):164–182, February 1993.
- [24] J. Zahorjan, E. Lazowska, and D. Eager. Spinning Verses Waiting in Parallel Systems with Uncertainty. In *Proceedings of the International Seminar on Distributed and Parallel Systems*, pages 455–472, December 1988.