Practical Considerations for Lock-Free Concurrent Objects

Brian N. Bershad
September 1991
CMU-CS-91-183 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

An important class of concurrent objects are those that are *lock-free*, that is, whose operations are not contained within mutually exclusive critical sections. A lock-free object can be accessed by many threads at a time, yet clever update protocols based on atomic Compare-And-Swap operations guarantee the object's consistency.

In this paper we take a practical look at the Compare-And-Swap operation in the context of contemporary shared memory multiprocessors. We first describe an operating system-based solution that permits the construction of a non-blocking Compare-And-Swap function on processor architectures that only support lock-oriented atomic primitives. We then evaluate several locking strategies that can be used to synthesize a Compare-And-Swap operation. We show that the common techniques for reducing the overhead of lock-oriented synchronization in the presence of contention are inappropriate when used as the basis for lock-free synchronization. We then describe a simple modification to an existing synchronization protocol which allows us to avoid much of the overhead normally associated with contention.

This research was sponsored in part by a National Science Foundation Presidential Young Investigator Award (PYI), the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035 and in part by the Open Software Foundation (OSF).

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, DARPA, OSF, or the U.S. government.

Keywords: Operating Systems, Mutual Exclusion, Performance

1 Introduction

Programs running on shared memory multiprocessors rely on low-level synchronization mechanisms and protocols to ensure controlled access to concurrent objects. An important class of concurrent objects are those that are *lock-free*, that is, whose operations are not contained within mutually exclusive critical sections. A lock-free object can be accessed by many threads at a time, yet clever update protocols based on atomic Compare-And-Swap operations guarantee the object's consistency. Because threads are not forced to queue while accessing a lock-free object, they are not vulnerable to the effects of scheduling convoys, priority inversion, and deadlock. Further, lock-free objects may be accessed concurrently, resulting in a higher throughput for operations.

Several researchers have demonstrated the feasibility of lock-free objects. Wing and Gong [Wing & Gong 90], and Mellor-Crummey [Mellor-Crummey 87] have designed a library of lock-free concurrent objects and proven them correct. Herlihy has shown several practical algorithms for wait-free — a restricted form of lock-free — objects [Herlihy 90]. Massalin and Pu have implemented an entire operating system kernel for shared memory multiprocessors using only lock-free objects [Massalin & Pu 91].

The applicability of these results has been predicated on the assumption that an atomic Compare-And-Swap instruction is available in hardware. Indeed, many of the papers describing new lock-free algorithms contain a plea to hardware designers to include an atomic Compare-And-Swap instruction in future architectures for shared memory multiprocessors.

In this paper, we consider various aspects of this plea. First, we assert that it has largely gone unheard. Few of today's processor architectures (and therefore shared memory multiprocessors) support an atomic Compare-And-Swap instruction. Second, we describe a software approach for implementing Compare-And-Swap with a "less universal" [Herlihy 88] primitive such as Test-And-Set. Our approach relies on a small amount of operating system support so as to not to lose the advantages of lock-free synchronization. We then turn to the issue of performance, and explore various synchronization techniques that can be used to implement lock-free concurrency primitives. We show that the established techniques for reducing synchronization overhead in the presence of contention are not appropriate for lock-free synchronization. We then describe a set of alternatives for implementing Compare-And-Swap on bus-based shared memory multiprocessors, and demonstrate one strategy which works well in the the presence of contention by reducing the frequency of synchronization. Our approach exploits a subtlety in the definition of Compare-And-Swap that allows us to "guess" that a synchronized Compare-And-Swap is likely to fail before doing the synchronization.

The remainder of this paper is structured as follows. In Section 2 we motivate this work by describing some desirable properties of lock-free synchronization, and show that few processors support it directly. In Section 3 we describe a simple operating system mechanism which can be used to build an atomic Compare-And-Swap function in software, but which permits the construction of objects which behave as though they are lock-free. In Section 4 we describe a set of synchronization alternatives for the software implementation and show that traditional high-performance locking mechanisms are inappropriate for Compare-And-Swap.

2 Properties of Lock-Free Concurrent Objects

A *lock-free* object is one that can be accessed by multiple threads concurrently without having to acquire a software lock, such as a mutex or a semaphore. In contrast, a *blocking* object is protected by a lock which a thread acquires before manipulating the object and releases when done. While the lock is held, no other thread may access the object. Because the lock serializes all access, blocking objects exhibit no concurrency. Furthermore, threads waiting on blocking objects are vulnerable to the untimely preemption of the lock holder.

Lock-free objects come in several flavors. A *non-blocking* object guarantees that threads will not block while trying to access the object. A *linearizable* [Herlihy & Wing 90] object is one which can be operated on concurrently by several threads, but which at all times appears to have been manipulated according to some legal (defined in terms of the object's semantics) sequential history. In this sense, linearizability is similar to the notion of serializability found in transaction systems. Operations can be interleaved as long as the interleaving is consistent with some linear ordering. A *wait-free* object is one which is linearizable, and for which all threads complete in a finite number of steps [Herlihy 91]. Finally, a *strongly* or *bounded* wait-free object is one which is wait-free, and for which the number of steps is bounded [Herlihy 91].

Lock-free objects of all flavors are attractive for several reasons. First, they are not vulnerable to convoy effects, priority inversion, or deadlock — all common problems in parallel systems. Convoying occurs when a thread which is holding a lock is descheduled, say due to its quantum expiring, a page fault or an interrupt, and other threads are forced to wait because a lock is held by a non-running thread. Priority inversion occurs when a low priority thread holds a lock needed by a high priority thread, but the low priority thread has been preempted by a thread of medium priority. The high priority thread cannot make progress because a medium priority thread is preventing the low priority thread from releasing its lock. Deadlock occurs when threads hold locks while waiting for locks held by other threads. Since there is no explicit locking with lock-free objects, these effects cannot occur.

Lock-free objects which are linearizable may also permit greater concurrency because semantically consistent (non-interfering) operations may execute in parallel. Further, linearizability is a *local* property, and is therefore independent of any underlying scheduling policy or interaction between objects. Locality improves the portability and modularity of large concurrent systems, and can simplify reasoning about concurrent objects. Because of this, a rich collection of proof techniques for linearizable objects have been developed in recent years [Herlihy & Wing 87].

At the heart of many lock-free concurrent algorithms lies the requirement for an atomic Compare-And-Swap instruction. In its simplest form, Compare-And-Swap takes three arguments: the address of a shared data item, an old value of the shared data item, and a new value. A thread reads a shared data value, computes a new value based on the read (and now old) value, and then tries to swap the old value with the new value. If the current value of the shared data item is equal to the old value, then it is replaced by the new value. If not equal, Compare-And-Swap returns a failure code and does not modify the shared data item. The failed compare indicates that the old value is "too old" because another thread had modified the shared data. The implication is that the new value is also invalid, since it is presumably computed based on the old value. In effect, the failing thread

discovers contention for shared data after the fact, forcing it to again read the shared data item's value and compute a new value. (A more general form of Compare-And-Swap is the Compare-And-Swap-N operation, which allows N separate locations to be atomically compared and swapped. Compare-And-Swap-N is helpful when implementing complicated shared data structures such as doubly-linked lists.)

Unfortunately, few contemporary processors support Compare-And-Swap directly. Instead, most support simpler atomic operations such as Test-And-Set or an Atomic Exchange. For example, of eight production-quality shared memory multiprocessors (Encore's Multimax, Sequent's Symmetry, SGI's MIPS-based multiprocessor, Omron's Luna88k, Sony's NEWS, DEC SRC's Firefly, and DEC's 6380 and 433MP Corollary), only two (the 486-based Corollary and the 68030-based NEWS) have a processor which implements Compare-And-Swap. Even when a Compare-And-Swap is part of the instruction set, it is not always the case that the memory system can support it. This is true, for example, with the 68020 processors on the Butterfly Plus. The lack of widespread hardware support for Compare-And-Swap means that it is necessary to find an efficient software solution to permit the implementation of lock-free concurrent algorithms.

3 Implementing Compare-And-Swap Without Direct Hardware Support

Many papers describing lock-free algorithms suggest that Compare-And-Swap is a critical hardware function that must be architecturally supported on shared-memory multiprocessors. This conclusion follows from the observation that a straightforward simulation of Compare-And-Swap using simpler primitives, such as is shown in Figure 1, would not make it possible to build lock-free objects. A thread which is preempted within the critical section would delay any other thread trying to perform the Compare-And-Swap for an unbounded amount of time. In contrast, direct hardware support for Compare-And-Swap provides a non-interruptible instruction that cannot be preempted.

```
int Compare_And_Swap(address, old_value, new_value)
        int *address;
        int old_value;
        int new_value;
{
    1               acquire_lock();          /* BEGIN CRITICAL SECTION */
    2               if (*address == old_value)       {
    3                       *address = new_value;
    4                       release_lock();        /* END CRITICAL SECTION */
    5                       return SUCCESS;
    6               } else  {
    7                       release_lock();        /* END CRITICAL SECTION */
    8                       return FAILURE;
    9               }
}
```

Figure 1: Implementing Compare-And-Swap With Locks

The fundamental problem with building Compare-And-Swap out of simpler primitives is caused by the operating system, which schedules threads preemptively. Specifically, the

critical section in the Compare-And-Swap sequence can be interrupted by the operating system, leaving other threads with no way of executing a Compare-And-Swap. In this section, we describe a solution that requires a small amount of operating system support to, appropriately, solve an operating system problem.[1]

We propose a recovery-based solution based on *roll-out*. With roll-out, a thread preempted within a Compare-And-Swap sequence causes the lock to be released immediately when the preemption occurs. Moreover, if the thread had not yet executed the swap, it is set to resume execution at the beginning of the sequence, otherwise it resumes at the end. For example, for the code in Figure 1, a thread preempted after line 1, but before executing either the store at line 3 or the lock release at line 7 would be scheduled to resume at line 1. If the thread was preempted after line 3 but before line 4, then the lock would be released and the thread would have to be rolled-out to resume at line 5.

Roll-out can be implemented with a small amount of operating system support. The machinery described in [Anderson et al. 90], for example, gives control back to the application at a fixed address immediately following the preemption. Code at that address can determine that the just preempted thread was executing in the Compare-and-Swap and can perform the necessary cleanup. A less general solution has the kernel performing the cleanup by discovering that the preempted thread was in a Compare-And-Swap sequence. This can be done through the use of code sequences at distinguished addresses, on-the-fly inspection of the code stream, or a designated per-thread variable which is toggled on entry and exit from the sequence.

The roll-out solution is similar to the notion of restartable atomic sequences used for implementing atomic operations on a uniprocessor [Bershad 91]. In that case, a critical sequence could be guaranteed to execute atomically *eventually* as long as a thread preempted within the sequence was restarted at the beginning. Roll-out for Compare-And-Swap is slightly less general because a lock must be released as well.

An alternative to roll-out is *roll-forward*. With roll-forward, the remaining code in the Compare-And-Swap sequence is executed and the lock is released at the point when the preemption occurs. Roll-forward has several problems, though, that make it less attractive than roll-out (which only requires that the lock be released and the thread's PC changed). Roll-forward requires executing code on behalf of a thread within the context of another thread. If the roll-forward is being handled in the kernel, then the kernel must be careful about any memory references it makes during the roll-forward to ensure that they are within the addressing domain of the stopped thread. More importantly, if the thread stopped because of a page fault, then it might not be possible to perform the roll-forward at all until the fault is satisfied.

With operating system support for Compare-And-Swap, the problems normally associated with lock-based synchronization do not occur. Indefinite convoying is impossible because a lock held by a preempted thread is released immediately after the preemption. Priority inversion is avoided because lower priority threads cannot hold locks indefinitely after being preempted by higher priority threads. Deadlock is avoided because it is not possible to execute arbitrary code while holding a lock. Lock acquisitions cannot be nested, so there can be no cycles in the "waits-for" relationship of threads.

[1] It's worth noting that a kernel-level solution, adequate for mainpulating operating system data structures from within the kernel, can be achieved through the simple raising and lowering of processor priorities.

One advantage of the software approach is that it allows for arbitrary generalizations of Compare-And-Swap. For example, the compare function need not just test for equality, but could test any kind of binary relation. Additionally, Compare-And-Swap could operate on objects that span multiple words. In contrast, Compare-And-Swap implemented in hardware is generally limited to a single word comparisons.[2] Herlihy described implementable non-blocking algorithms in which that single word was a pointer to the actual shared data [Herlihy 90]. Much of the complexity and cost of his algorithms, however, was due to the overhead of having to use pointers and manage memory. This complexity disappears with a multiword Compare-And-Swap (although the implementation of roll-out does become more complicated).

4 Locking Strategies for Lock-Free Synchronization

While operating system support can guarantee the progress of a synthesized Compare-And-Swap operation, proper locking strategies are necessary to ensure good performance. This is especially true during periods of high contention for shared data objects.

In this section, we consider a set of different strategies for implementing the `acquire_lock` operation from Figure 1. We begin by demonstrating that a straightforward implementation based on spinlocks performs poorly even in the presence of small amounts of contention. We then show that software queueing [Anderson 90, Graunke & Thakkar 90, Mellor-Crummey & Scott 91], a locking strategy designed to perform well in the presence of contention, is inappropriate for synchronizing with Compare-And-Swap. Finally, we evaluate strategies that allow the Compare-And-Swap to non-atomically and conservatively fail. Our intention is to reduce the frequency with which Compare-And-Swap requires an expensive atomic operation only to end in failure.

4.1 Hardware Platforms

We use two successive generations of shared memory multiprocessor architectures to evaluate the various locking strategies. Both are bus-based, cache coherent, and use a write-invalidate coherency protocol. A Sequent Symmetry with 20 Intel 386 processors running at 16.67 Mhz represents the older generation. The newer generation is an Omron Luna88k multiprocessor workstation with 4 Motorola 88100 processors running at 25 Mhz. Neither supports a Compare-And-Swap operation directly in hardware. The Symmetry and the Luna88k each have an instruction that allows a register and a memory location to be atomically swapped.

Single-bus-based shared memory multiprocessors use the system bus as an arbitration mechanism. A processor performs an atomic operation by asserting a special signal on the bus. This prevents other processors from performing atomic operations until the signal is removed. On systems that use a write-invalidate protocol, the special signal can also cause other copies of the synchronization data to be invalidated. In all cases, however, the atomic operation involves at least one bus transaction. On systems with write-through caches, or

[2]The Compare-And-Swap-2 operation found on the 680x0 series could be used to implement a 64 bit Compare-And-Swap.

on systems which require that synchronization operations be performed to memory, the modification can involve an additional bus and memory transaction.

The two generations of multiprocessors help to illustrate that the relative cost of performing atomic operations has increased substantially with processor speed. This is because of the growing imbalance between processor speed and bus and memory speed. The Intel 386 in the Sequent Symmetry takes about twice as long to execute an atomic exchange to a cached memory location as it does to increment it. The Motorola 88000, which is a RISC-based microprocessor takes about 6 times longer for the atomic exchange. At the extreme end of the spectrum are machines like Stanford's DASH multiprocessor. The processors in DASH, based on the MIPS R3000, take about sixty times longer to synchronize as they do to execute simple instructions [Lenoski et al. 90].

Two points are implied by these trends. First, it is becoming increasingly important to reduce the frequency of unnecessary synchronization because its cost is no longer negligible. We will see later in this section how a straightforward implementation of Compare-And-Swap can involve a large number of unnecessary synchronizations. Second, because synchronization operations run relatively more slowly on faster processors, the synchronized component of a lock-free object will tend to dominate performance sooner on faster processors than on slower ones. For example, code that implements a lock-free data structure, such as a linked list, efficiently on a older generation shared memory multiprocessor, may not be efficient when executed on a newer machine because of the divergence in the relative performance of synchronizing and non-synchronizing operations.

4.2 Measuring Performance

We use throughput as the primary measurement for evaluating the performance of synchronization strategies for lock-free concurrent objects. We compute throughput by having a fixed number of processors execute a loop which contains a Compare-And-Swap. The code executes for a fixed period of time.

```
1:  while (should_stop == FALSE)        {
2:        do  {
3:                old_value = shared_data;
4:                new_value = compute_new_value(old_value);
5:                res = Compare_And_Swap(&shared_data,  old_value, new_value);
6:        } while (res != SUCCESS);
7:        success[me] = success[me] + 1;
8:  }
```

Figure 2: Code Loop for Measuring Throughput

The code for the loop is shown in Figure 2. The variable `should_stop` is set by a special thread which wakes when a timer expires. The array `success` is used to keep track of the number of times that each thread is able to successfully update the variable `shared_data` (we use a per-thread data structure for collecting statistics to avoid extra locking and contention). Throughput, then, is simply the total number of successes that occurred during the test.

Clearly, if the function `compute_new_value` takes a long time relative to the Compare-And-Swap, then the impact of the Compare-And-Swap on throughput is going to be small,

since threads will be executing non-synchronized code most of the time. As the time to execute `compute_new_value` decreases relative to the Compare-And-Swap, the effects of synchronization will begin to dominate. For the measurements presented in this paper, the function `compute_new_value` is implemented as a loop which cycles for a fixed number of times, and returns a different value for each thread. An "execution time" of w corresponds to w passes through the loop.

Throughput measurements with one processor reveal the basic latency of the Compare-And-Swap operation when paired with the function `compute_new_value`. Throughput measurements with many processors illustrate their behavior in the presence of contention when many threads try to access the `shared_data` variable at the same time.

Ideally, the rate of synchronization operations for Compare-And-Swap should be equal to the throughput, that is, the rate of successful compares followed by a swap. When concurrent objects are built using traditional locks and critical sections, each synchronization operation is followed by a successful access operation, so synchronization operations are never "wasted." An implementation for Compare-And-Swap should exhibit the same property.

4.3 Simple Spin Locks

In this section we examine the effect on throughput of a synchronization policy based on spinlocks. We implement the `acquire_lock` operation directly using an atomic Test-And-Set operation such as the one shown in Figure 3. The function Test-And-Set atomically sets a memory location and returns its previous value. The non-destructive read loop before the Test-And-Set creates a Test-And-Test-And-Set operation, which allows a processor to read-spin on cached value of the lock, rather than to generate bus activity during each pass through the spin loop. This is a common spinlock optimization [Rudolph & Segall 84].

```
acquire_lock()
{
   while (1)  {
          while ( lock != 0 )          /* wait until lock is free */
                 ;
          if (Test_And_Set(lock) == 0)
                 return;
    }
}
```

Figure 3: Simple Test-And-Test-And-Set Spinlock

The absolute throughput for the code in Figure 2 is shown in Figure 4. Throughput for both multiprocessors is shown. The numbers in the legend reflect runs using different values of w. The x-axis graphs number of processors and the y-axis graphs the total number of successful operations. Each graph contains a family of curves, where each curve represents a different "compute" time, ranging from $w = 1$ to $w = 1000$. The $w = 1$ case corresponds to a "worst case" ratio of compute to synchronization time, whereas $w = 1000$ corresponds to the case where compute time dominates synchronization time.

As expected, smaller compute times result in higher throughput because it takes less time to make it through one pass of the code in Figure 2. Except for the $w = 1$ case on the Luna88k, throughput increases slightly as processors are added and then drops off. The

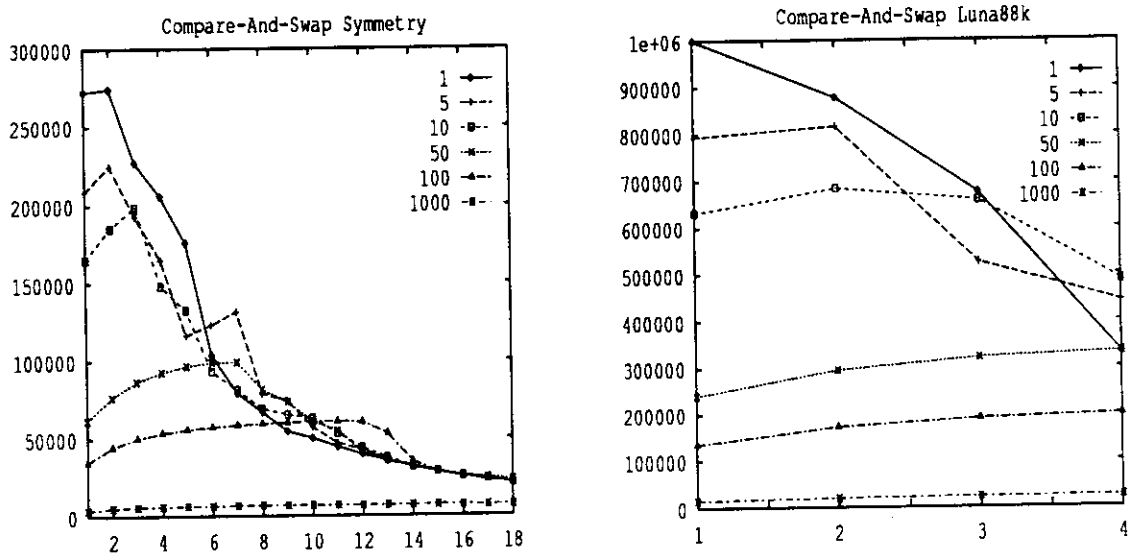

Figure 4: Compare-And-Swap using spinlocks.

small initial rise in throughput is due to the fact that not all of the code in the measured loop is sequential. In particular, the code at lines 1,2,3,6,7 and 8 in Figure 2 can usefully execute in parallel. The improvement due to this parallelism is offset by the increased overhead of lock and data contention that comes with more processors. This is why the curves for smaller w turn down more quickly than those for larger w. At small w, lock and data contention are high, therefore the benefit due to the parallelism in the loop disappears quickly as processors are added (and doesn't exist at all on the Luna88k when $w = 1$). When w is large, however, lock and data contention are reduced so the beneficial effect of the loop's parallelism takes longer to undermine.

The graphs also illustrate that throughput drops off more rapidly on Luna88k's faster processors where the cost of synchronization and bus access are much higher relative to the Symmetry. For example, the $w = 1$ case shows a factor of 3 reduction in throughput at four processors on the Luna88k, whereas the reduction is only a factor of 1.3 on the Symmetry.

At least two effects are responsible for the rapid dropoff in throughput. First, there is the commonly observed degradation that occurs when many threads try to synchronize. Although threads spinwait on a cached value of the lock, the release of the lock is broadcast to all waiting processors. Each then tries to reacquire the lock. Although one will succeed, the others will execute a synchronizing Test-And-Set, placing a load on the bus.

A second reason for the slowdown is that a failed compare incurs a synchronization cost which affects all processors, but which contributes nothing to total throughput. We can factor out synchronization and failure effects and just look at behavior due to the locking protocol by modifying the loop so that each thread does a Compare-And-Swap on a different memory location. In this way, every Compare-And-Swap succeeds and there is no bus contention due to keeping shared data consistent. Threads interact only because they use the same lock to gain access to the Compare-And-Swap sequence. The resulting

curves are shown shown in Figure 5. As the number of processors increases, throughput first increases linearly and then drops off.

Because only part of the loop is sequential, increasing the number of processors also increases throughput. Eventually, though, the sequential Compare-And-Swap limits throughput. For smaller compute times the limit is reached with fewer processors because most of the code is serial. Throughput then drops off because of the bus contention that arises when many processors simultaneously vie for the same spinlock. There is a flurry of bus activity when the spinlock is released, effectively slowing down all processors. The behavior demonstrated in Figure 5 closely matches that observed by Anderson, and Graunke and Thakkar.

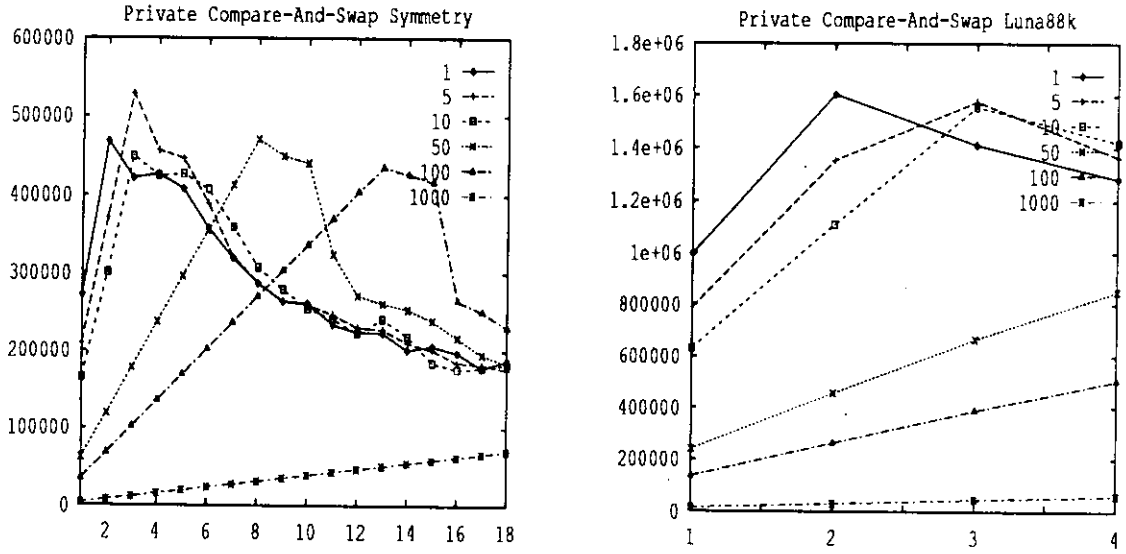

Figure 5: Compare-And-Swap using spinlocks. Each thread accesses a different location.

The effect of failure with non-blocking mechanisms contradicts a common belief that a non-blocking synchronization protocol should be no worse in the case of contention than one that blocks. In both cases cycles are wasted as threads wait for one thread to finish an inherently serial code sequence. Unfortunately, the straightforward implementation of a non-blocking mechanism causes cycles to be lost on all processors, not just those that are waiting, whenever there is contention.

4.4 Contention-Tolerant Locking with Software Queueing

The degradation in throughput shown in Figures 4 and 5 suggests that a locking strategy which reduces bus contention due to synchronization might improve performance. In this subsection, we consider behavior when using *queuelocks* [Anderson 90, Graunke & Thakkar 90, Mellor-Crummey & Scott 91] to guard the Compare-And-Swap operation.

The idea behind queuelocks is that each thread waits only for one other thread to release the lock. For example, the first waiting thread is waiting for the actual lock holder, and the second waiting thread is waiting for the first waiting thread. This relationship permits each thread to spin on a different memory location. A thread releases a lock by

depositing a new value into the memory location associated with the next waiting thread. The result is low bus contention even in the presence of high lock contention because the number of synchronization operations equals the number of successful synchronizing bus operations. Queuelocks have been shown to be effective at reducing bus contention, and at maintaining near constant throughput out to large numbers of processors for algorithms that use traditional lock-based synchronization.

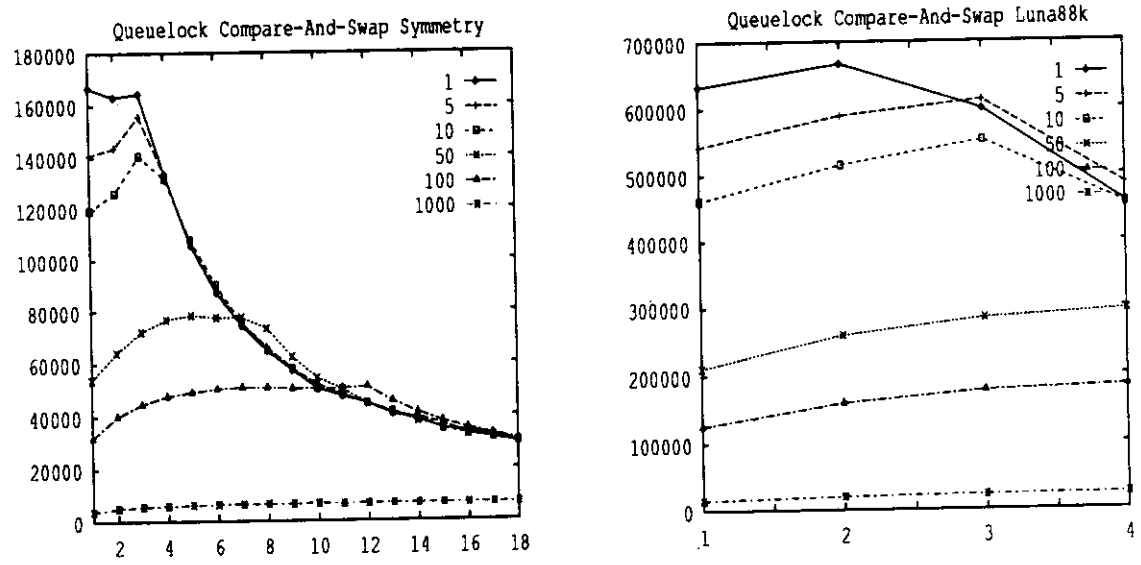

Figure 6: Compare-And-Swap using queuelocks.

We use queuelocks to implement the `acquire_lock` function from Figure 2. The measured throughput is shown in Figure 6. By comparing the curves to those in Figure 4, several things are apparent. First, at low contention, queuelocks have lower throughput than spinlocks because they have a more complex implementation (our implementation follows Graunke's and Thakkar's). Relative to the spinlock solution with one processor, for example, throughput with queuelocks is reduced by factor of 2. More importantly, however, the performance profile for queuelocks is not that much different than for spinlocks. There is a slight rise in throughput at small numbers of processors, and then a dropoff as the number of processors increases. This behavior contradicts that seen when queue locks are used to manage critical sections for lock-based concurrent objects.

The dropoff in throughput is *not* due to synchronization overhead, which queuelocks eliminate, but to the fact that threads must delay before executing the Compare-And-Swap. When a thread waits on a queuelock, it delays until all threads ahead of it acquire the queuelock, attempt the Compare-And-Swap and then release the lock. If the waiting thread succeeds, then the threads waiting ahead of it must have failed. In the worst case, with n threads queued, a successful Compare-And-Swap can be performed only once every nt cycles, where it takes time t to execute the critical section. This is because each succeeding thread must delay behind $n-1$ failing threads, each of which takes time t to discover their failure. (We ignore here the beneficial effects of the loop's parallelism, which account for an initial rise in throughput at low numbers of processors.)

We can separate the effects of failure from those of synchronization by again having each thread do its Compare-And-Swap to a different memory location. In this case, which is shown in Figure 7, throughput does not drop off as the number of processors increases. Bus contention due to synchronization is minimized and every Compare-And-Swap contributes to total throughput. Unfortunately, real concurrent objects require updates to common data, so this benchmark is only useful for understanding the effects of queueing and failure.

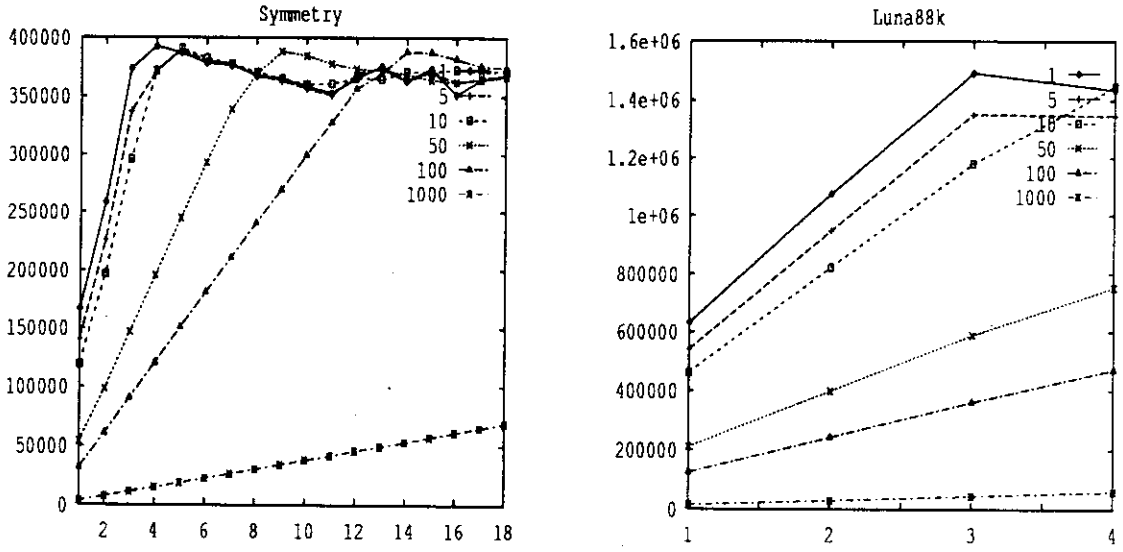

Figure 7: Compare-And-Swap using queuelocks. Each thread accesses a different location.

4.5 Minimizing Synchronization with Conservative Compare-And-Swap

The problem with Compare-And-Swap based on spinlocks and queuelocks is that threads go through a global synchronization protocol only to then fail by discovering data contention. The total number of successful accesses is ultimately bounded by the time to execute the Compare-And-Swap sequence itself. As more processors are added, however, the total number of attempts increases. Since the number of successes is bounded, this results in an increase in the number of failures. Because failures have a non-local cost (in terms of bus synchronization and queueing delay), throughput drops off.

In this subsection we examine several methods for building a Compare-And-Swap operation which attempt to reduce the cost of failure. Our approach is to change the definition of Compare-And-Swap so that it becomes *advisory*. A failure returned from Compare-And-Swap only means that no swap occurred, but not necessarily that the old value and the current value are different. This enables a Compare-And-Swap to fail before having to execute an expensive synchronization operation. Instead, we can determine if the Compare-And-Swap is likely to fail and go no further.

Compare-And-Compare-And-Swap

The first strategy, Compare-And-Compare-And-Swap, compares the old value with the current value before executing the `acquire_lock` operation. If the two values are equal, the lock is acquired, and the Compare-And-Swap is actually performed. The initial compare is intended to address the problem of a thread queueing to acquire a lock only then to discover that its old value is obsolete.

Figure 8 shows throughput for increasing numbers of processors on the Luna88k and the Symmetry. Surprisingly, the graphs show behavior that is quite similar to that for Compare-And-Swap. Throughput increases at first and then drops off as more processors are added with the dropoff coming with smaller numbers of processors for shorter compute times.

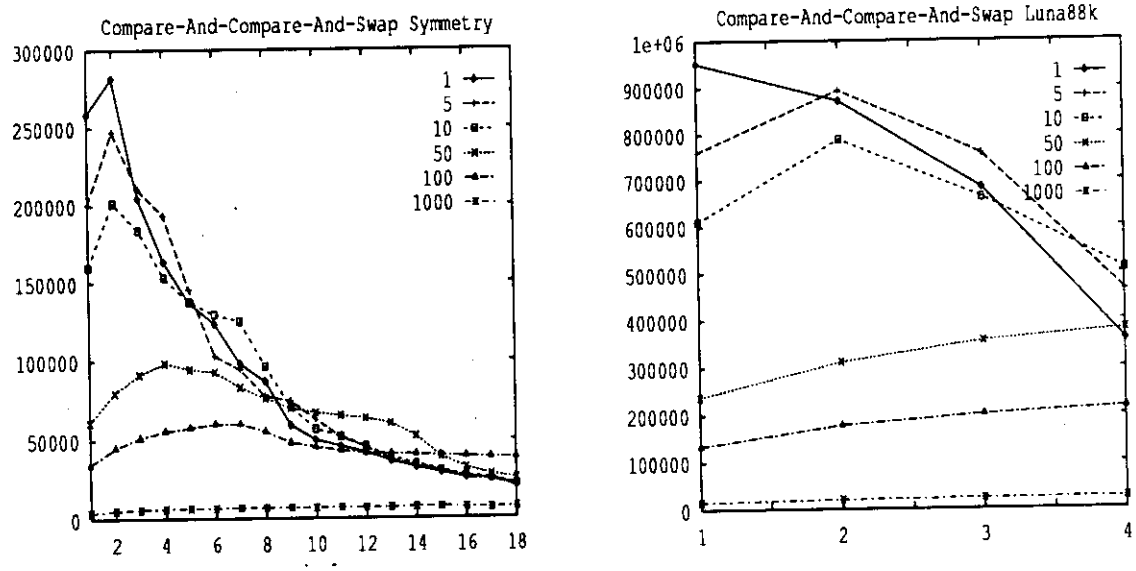

Figure 8: Compare-And-Compare-And-Swap using spinlocks.

The explanation for the unexpected behavior in Figure 8 is somewhat subtle. When the compute time w is large, contention is naturally low. The extra check therefore makes little difference because it only eliminates a synchronization operation that would have occurred during periods of low contention, when synchronization is cheap. On the other hand, when w is small, there is only a small chance that a thread will read the shared variable, compute, and then find a new value with its first compare in the Compare-And-Compare-And-Swap. Consequently, the thread goes through the process of acquiring the lock. But, also because the compute time is short, contention for the shared variable is high, and there is a good chance that the thread finds other threads accessing the lock. This gives rise to the following behavior: a thread does an initial read of the shared data, computes for a short time, rereads the shared data and finds it unchanged, delays while acquiring the lock, acquires the lock (which takes a long time), finds that the shared data has changed during the lock delay, and then releases the lock. As a result, the additional compare with Compare-And-Compare-And-Swap isn't very useful.

Compare*-And-Compare-And-Swap

We can improve upon the effectiveness of the single initial compare by polling for equality while a processor waits for the lock. A waiting processor can therefore abort the lock acquisition and the Compare-And-Swap altogether if it detects that the shared value and the old value are not equal. This has the effect of purging from the set of waiting processors those processors whose Compare-And-Swap will most probably fail when it ultimately occurs. Successful processors therefore only have to wait for other successful processors; failure incurs no global queueing delay.

The throughput of this strategy, which we call Compare*-And-Compare-And-Swap, is shown in Figure 9. The graphs were generated using Test-And-Test-And-Set spinlocks in which the initial test loop also included a check for equality. Compared to the previous techniques, the additional compare substantially reduces the rate at which throughput degrades when processors are added. Moreover, its absolute performance in the low contention cases is comparable to straightforward Compare-And-Swap.

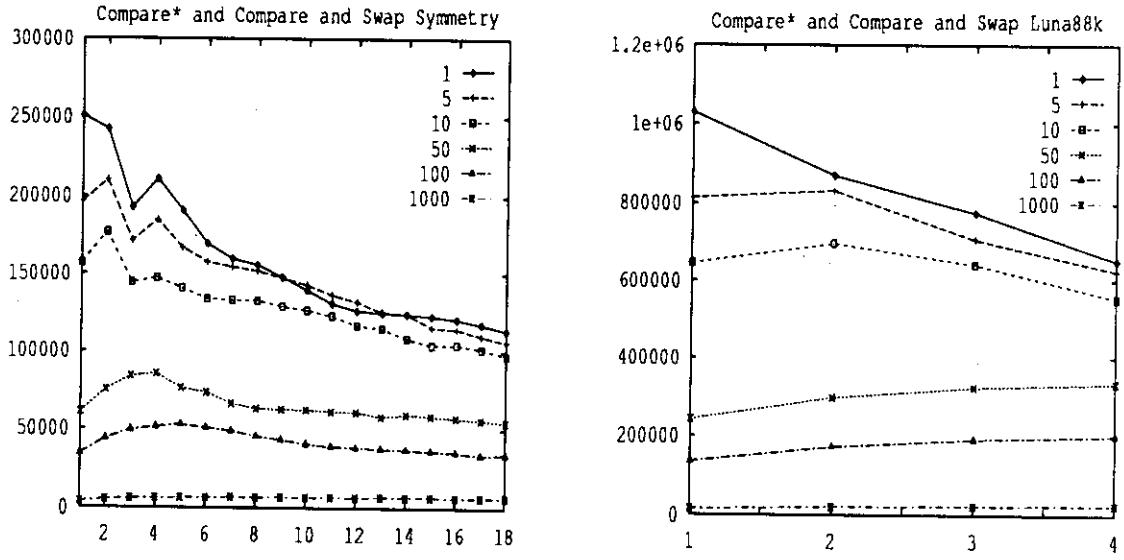

Figure 9: Compare*-And-Compare-And-Swap

4.6 Summary

Synchronization protocols appropriate for lock-based concurrent objects are inappropriate in terms of performance when used as the basis for a lock-free concurrent objects. In the presence of contention, simple spinlocks generate excessive bus contention. Queuelocks, which are highly effective for reducing contention with lock-based concurrent objects, are not effective when used with lock-free concurrent objects. They create a situation in which threads queue on the lock only to fail on the compare. Throughput decreases as processors are added because the number of failed Compare-And-Swaps grows. We can eliminate the effect of failure on queueing delay and therefore throughput by prematurely aborting the Compare-And-Swap if the shared value and old value become unequal while trying to

acquire the lock.

5 Conclusions

Lock-free concurrency has the potential to become a powerful and efficient model for shared memory parallel programs. To date, most work in the area has been theoretical, or of the "proof of concept" style. In this paper, we have explored several practical considerations for systems that rely on lock-free concurrency. We have described a simple operating system mechanism with which one can build "practically" lock-free constructs out of lock-based ones. We then described a set of synchronization policies for implementing the lock-free mechanisms, and have shown that throughput can be extremely vulnerable to contention. Specifically, we have shown that it is the cost of failure with Compare-And-Swap that can have the greatest effect on overall throughput when contention is high. We have shown that it is possible to reduce this cost by slightly relaxing the definition of Compare-And-Swap so that failures can occur outside the normal atomic protocol.

Acknowledgements

I'd like to thank Dan Stodolsky for his help in evaluating the tradeoffs between various synchronization policies. He, in addition to Greg Morrisett and Steve Schwab, provided valuable feedback on this paper's prose and presentation. Jeannette Wing helped me to better understand the subtleties and benefits of the different styles of lock-free concurrency. A conversation with Maurice Herlihy help me convince myself that the ideas in this paper were worth pursuing.

References

[Anderson 90] Anderson, T. E. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[Anderson et al. 90] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. Technical Report 90-04-02, Department of Computer Science and Engineering, University of Washington, April 1990. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991. To appear, *ACM Transactions on Computer Systems 10*(1), February 1992.

[Bershad 91] Bershad, B. N. Mutual Exclusion for Uniprocessors. Technical Report CMU-CS-91-116, School of Computer Science, Carnegie Mellon University, April 1991.

[Graunke & Thakkar 90] Graunke, G. and Thakkar, S. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[Herlihy & Wing 87] Herlihy, M. P. and Wing, J. M. Axioms for Concurrent Objects. In *Fourtheenth ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[Herlihy & Wing 90] Herlihy, M. P. and Wing, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Herlihy 88] Herlihy, M. Impossibility and Universality Results for Wait-Free Synchronization. In *Seventh ACM Symposium on Principles of Distributed Computing (PODC)*, pages 276–290, August 1988.

[Herlihy 90] Herlihy, M. A Methodology for Implementing Highly Concurrent Data Structures. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 197–206, March 1990.

[Herlihy 91] Herlihy, M. Wait-free Synchronization. *ACM Transactions on Programming Languages*, 13(1), January 1991.

[Lenoski et al. 90] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, May 1990.

[Massalin & Pu 91] Massalin, H. and Pu, C. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, 1991.

[Mellor-Crummey & Scott 91] Mellor-Crummey, J. M. and Scott, M. L. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1), February 1991.

[Mellor-Crummey 87] Mellor-Crummey, J. M. Concurrent Queues: Practical Fetch-and-ϕ Algorithms. Technical Report 229, Department of Computer Science, University of Rochester, November 1987.

[Rudolph & Segall 84] Rudolph, L. and Segall, Z. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 340–347, 1984.

[Wing & Gong 90] Wing, J. M. and Gong, C. A Library of Concurrent Objects and Their Proofs of Correctness. Technical Report CMU-CS-90-151, School of Computer Science, Carnegie Mellon University, July 1990.