Practical Considerations for Non-Blocking Concurrent Objects

A version of this paper appears in the May 1993 Distributed Computing Systems Conference (DCS).

Brian N. Bershad
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Brian.Bershad@CS.CMU.EDU

Abstract

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

In this paper we take a practical look at the Compare-And-Swap operation in the context of contemporary bus-based shared memory multiprocessors, although our results generalize to distributed shared memory multiprocessors. We first describe an operating system-based solution that permits the construction of a non-blocking Compare-And-Swap function on architectures that only support more primitive atomic primitives such as Test-And-Set or Atomic Exchange. We then evaluate several locking strategies that can be used to synthesize a Compare-And-Swap operation, and show that the common techniques for reducing synchronization overhead in the presence of contention are inappropriate when used as the basis for non-blocking synchronization. We then describe a simple synchronization strategy that has good performance because it avoids much of the synchronization overhead that normally occurs when there is contention.

This research was sponsored in part by a National Science Foundation Presidential Young Investigator Award, the Digital Equipment Corporation, 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 MDA972-90-C-0035, and 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, DEC, DARPA, OSF, or the U.S. government.

1 Introduction

Programs running on 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 *non-blocking*, that is, whose operations are not contained within mutually exclusive critical sections [Herlihy & Wing 90, Herlihy 91]. A non-blocking object can be accessed by many processors at a time, yet update protocols that use an atomic Compare-And-Swap guarantee the object's consistency. In contrast, a *blocking* object serializes access within critical sections where locks are used to control access.

Non-blocking concurrent objects have several qualities that make them attractive for parallel processing. Because processors are not forced to queue while accessing a non-blocking object, they are not vulnerable to the effects of scheduling convoys, priority inversion, and deadlock — all potential problems in parallel systems [Zahorjan et al. 88]. Convoying occurs when a thread is descheduled, say due to its quantum expiring, a page fault or an interrupt, forcing other processors to wait because the descheduled thread is holding a lock. 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 the medium priority thread is preventing the low priority thread from releasing its lock. Deadlock occurs when processors hold locks while waiting for locks held by other processors. All of these problems occur because processors block waiting for a resource. Since there is no blocking with non-blocking objects, their occurrence is prevented.

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

In practice, two problems must be addressed by the implementors of non-blocking objects. First, few contemporary processors support Compare-And-Swap directly. Instead, most support simpler atomic operations such as Test-And-Set or 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 that implements Compare-And-Swap.

The second problem with non-blocking algorithms is their performance in the presence of contention. As the number of processors concurrently accessing a non-blocking object increases, there exists the danger of degraded throughput due to the cost of global synchronization operations. This degradation is similar to that observed in the synchronization behavior of blocking objects [Anderson 90, Graunke & Thakkar 90, Mellor-Crummey & Scott 91].

In this paper we present practical solutions to these two problems. In Section 2 we describe an efficient software mechanism that can be used to build non-blocking concurrent objects. Our approach transforms a simple Test-And-Set operation into a non-blocking Compare-And-Swap using a small amount of operating system support. In Section 3 we describe a set of synchronization policies for the software implementation, and show that the established techniques for reducing synchronization overhead in the presence of contention are not appropriate for use in non-blocking algorithms on multiprocessors. We then describe a policy that works well in the presence of contention by relaxing slightly the definition of Compare-And-Swap so that it can conservatively fail. This has the effect of reducing the frequency of global synchronization, which is necessary for good performance in the presence of contention. In Section 4 we discuss related work. We present our conclusions in Section 5.

2 Compare-And-Swap

At the heart of many non-blocking concurrent algorithms lies 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 processor 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 processor had modified the shared data. The implication is that the new value is also invalid, since it presumably had been computed based on the old value. In effect, the failing processor discovers contention for shared data after the fact, allowing it to reread the shared data item's value and compute a new value.[1]

The Compare-And-Swap must be both atomic and non-blocking. It must also be *available*. As mentioned in the introduction, though, few contemporary shared memory multiprocessors support a Compare-And-Swap instruction. In the absence of an explicit hardware instruction, it must be synthesized using other available synchronization primitives. Unfortunately, a straightforward simulation of Compare-And-Swap using simpler primitives such as Test-And-Set to implement a mutually exclusive lock as shown in Figure 1 is not non-blocking. The operating system can preempt a thread within the critical section, delaying other threads for a possibly unbounded length of time.

```
int Compare_And_Swap(address, old_value, new_value)
        int *address;
        int old_value;
        int new_value;
{
  acquire_lock();   /* BEGIN CRITICAL SECTION */
  if (*address == old_value)       {
     *address = new_value;
     release_lock();  /* END CRITICAL SECTION */
     return SUCCESS;
  } else  {
     release_lock();  /* END CRITICAL SECTION */
     return FAILURE;
  }
}
```

Figure 1: *Implementing Compare-And-Swap With Locks*

2.1 Implementing Compare-And-Swap without direct hardware support

The problem with building Compare-And-Swap from primitives oriented towards blocking synchronization, such as Test-And-Set, is caused by the operating system which schedules threads preemptively. We can solve this problem using *roll-out*. With roll-out, a thread can be preempted within the Compare-And-Swap critical section, but the lock is released at

[1] 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.

the time of preemption. If the thread had not yet executed the swap, it resumes at the beginning of the sequence, otherwise it resumes at the end. For example, in 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 resumed at line 1. If the thread was preempted after executing line 3 but before releasing the lock at line 4, it would be rolled-out to resume at line 5. We assume here that single instructions execute atomically (interrupts are precise), meaning that a thread is never preempted *while* executing an instruction. The preemption comes entirely after one instruction and before the next.

Roll-out can be implemented with operating system support. The machinery described in [Anderson et al. 92], for example, which gives control back to the application at a fixed address immediately following the preemption, can be used. Code at that address can determine that the just preempted thread was executing a Compare-and-Swap and can then perform the necessary cleanup. Alternatively, the kernel can perform 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 that is toggled on entry and exit from the sequence.

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 processors running 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 processors.

2.2 Alternatives and advantages

An alternative to roll-out is *roll-forward.* With roll-forward, at the point when the preemption occurs, the remaining code in the Compare-And-Swap sequence is executed and the lock is released. 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 instruction counter be changed). Roll-forward requires executing code on behalf of a thread within the context of another thread. If 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. Page faults are another problem with roll-forward; 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.

One advantage of a software approach to atomicity is that it allows for arbitrary generalizations of Compare-And-Swap. For example, the compare function can test any kind of binary relation, not just equality. Additionally, software Compare-And-Swap can operate on objects that span multiple words. In contrast, Compare-And-Swap implemented in hardware is generally limited to single word operations.[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 multi-word Compare-And-Swap (although the implementation of roll-out does become more complicated).

2.3 Drawbacks with the solution

Roll-out is a practical technique for implementing non-blocking concurrent objects. It does have two drawbacks, though, that could make it inappropriate for use in some settings. First, it assumes that processors are non-stop. If a processor fails in the middle of a synthesized Compare-And-Swap, then the roll-out won't occur, and the lock will remain held. A non-blocking object implemented with a "pure" Compare-And-Swap instruction can be immune to processor failure. In practice, however, very little else in a multiprocessor actually is, so this should not be a practical deficiency.

Pathological scheduling can create a more acute problem since roll-out does not guarantee that a processor makes forward progress. Specifically, a processor preempted at line 2 in Figure 1 is backed up to line 1. If the operating system quantum is just a few cycles, it would be possible for all processors to never make any forward progress. This situation would prevent the implementation of wait-free concurrent objects, which guarantee that some processor makes progress within a bounded number of steps. In practice, scheduling quantums are much longer than a few cycles, so we do not perceive this to be a serious practical problem. If it were to become one, then roll-forward, rather than roll-back, would be more appropriate.

3 Locking strategies

While operating system support can guarantee the "non-blockingness" of a synthesized Compare-And-Swap operation, proper locking strategies are neces-

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

sary to ensure good performance. This is especially true during periods of high contention for shared data objects.

In this section we examine 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 use with Compare-And-Swap. We then describe a strategy that allows Compare-And-Swap to conservatively and inexpensively fail. Our goal is to reduce the frequency with which Compare-And-Swap fails after requiring an expensive atomic operation – a pattern that occurs commonly during periods of high contention.

3.1 Hardware platforms

We use two successive generations of shared memory multiprocessor architectures to evaluate synchronization strategies for Compare-And-Swap. Both are bus-based, cache coherent, and use a write-invalidate coherency protocol. The older generation is represented by a Sequent Symmetry with 20 Intel 386 processors running at 16.67 Mhz. The newer generation is represented by 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. This is sufficient for implementing a synchronizing lock operation.

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 invalidate data. In all cases, however, the atomic operation involves at least one bus transaction. On systems with write-through caches, or on systems requiring that synchronization operations be performed through to memory, the modification can involve an additional bus and memory transaction.

Using two generations of multiprocessors helps to illustrate that the relative cost of performing atomic operations is increasing substantially with processor speed. This is because of the growing imbalance between processor speed, bus speed, 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 a cached memory location. In contrast, the Motorola 88100, a RISC-based microprocessor, takes about 6 times longer for the atomic exchange than for the increment. At the far end of the spectrum are machines like Stanford's DASH multiprocessor; its processors take about sixty times longer to synchronize using a shared memory location as they do to execute simple instructions [Lenoski et al. 90].

Two points are implied by this trend. First, it is becoming increasingly important to reduce the frequency of synchronization because its cost is no longer negligible. We will see later in this section that 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 non-blocking object will have a greater effect on performance on faster processors than on slower ones. For example, code that implements a non-blocking data structure efficiently on older generation shared memory multiprocessors, will be less efficient when executed on a newer machine because of the divergence in the relative performance of synchronizing and non-synchronizing operations.

3.2 Measuring the performance of synchronization alternatives

We use throughput as the primary measurement for evaluating the performance of synchronization strategies for non-blocking concurrent objects. We compute throughput by having a fixed number of processors execute a loop that contains a Compare-And-Swap. The code, which executes for five seconds (wall-clock), is shown in Figure 2. The variable `should_stop` is set by a special thread that wakes when a timer expires. The array `success` is used to keep track of the number of times that each thread successfully updates the variable `shared_data` (we use a per-thread data structure for collecting statistics to avoid extra locking and contention). Throughput is simply the total number of successes that occur during the test.

```
while (should_stop == FALSE)        {
   do  {
        old_value = shared_data;
        new_value = compute_new_value(old_value);
        res = Compare_And_Swap(&shared_data,
                               old_value, new_value);
   } while (res != SUCCESS);
   success[me] = success[me] + 1;
}
```

Figure 2: *Code loop for measuring throughput*

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 will be small, since processors will be executing non-synchronizing 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 that 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 reveal behavior in the presence of contention.

3.3 Simple spinlocks

We first examine the effect of contention on throughput when the `acquire_lock` operation is implemented using spinlocks. Spinlocks, in turn, are implemented using an atomic Test-And-Set operation (Figure 3) built from the atomic exchange provided by the processor. 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 that allows a processor to read-spin on a cached value of the lock, rather than 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*

Throughput for the code in Figure 2 on both multiprocessors is shown in Figure 4. The x-axis represents the number of processors and the y-axis represents the total number of successful operations that occurred. We ran our experiments out to 18 of the Sequent's 20 processors as that offered a wide enough range without introducing background activity into the experiment. A small amount of background activity is included, though, in the Omron experiments. 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 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 small initial rise in throughput is due to the fact that not all of the code in the measured loop is strictly sequential. In particular, the code at lines 1,2,3,6,7 and 8 in Figure 2 can 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. As a result, 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 the Luna88k's faster processors where the cost of synchronization is 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 simultaneously 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 that affects all processors, but that 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 in Figure 5. (The scale of the y-axis is different than than in the previous figure for reasons of resolution.) 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 compete for the same spinlock. There is a flurry of bus activity when the spinlock is released, effectively

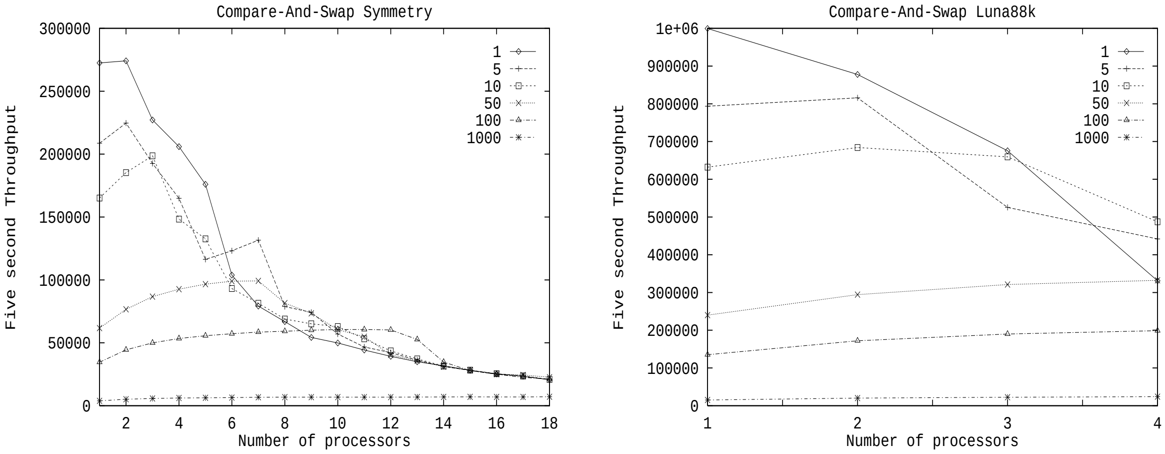

Figure 4: *Compare-And-Swap using spinlocks. The numbers in the legend reflect runs using different values of w. The graphs for both system configurations clearly show that throughput drops off significantly under contention when the compute time is small.*

slowing down all processors. The behavior demonstrated in Figure 5 closely matches that observed by Anderson, and Graunke and Thakkar in their studies of spinlock performance.

3.4 Contention-tolerant locking with software queueing

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

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. With queuelocks, each processor waits for one other processor to release the lock, rather than waiting for the lock itself. For example, the first waiting processor would wait for the actual lock holder to release the lock, while the second waiting processor would wait for the first waiting processor to acquire and then release the lock. This relationship permits each processor to poll a different memory location. A processor releases a lock by depositing a new value into the memory location associated with the next waiting thread. Queuelocks reduce bus contention because the number of synchronization operations equals the number of successful synchronizing bus operations, even in the presence of high lock contention.

Figure 6 shows throughput when queuelocks are used to implement the `acquire_lock` function from Figure 2. By comparing the curves to those in Figure 4, several things become apparent. First, at low contention, queuelocks have lower throughput because they are more complicated to implement (our implementation follows that of Graunke and Thakkar). 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 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 queuelocks 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 processors delay before executing the Compare-And-Swap. When a processor waits on a queuelock, it delays until all processors ahead of it acquire the queuelock, execute the Compare-And-Swap, and release the lock. If a waiting processor succeeds, then the processors that were waiting ahead of it must have failed, and those waiting behind will fail. Specifically, with n processors queued, a successful Compare-And-Swap can be performed only once every nt cycles, where the Compare-And-Swap takes t cycles. (We ignore the beneficial effects of the loop's parallelism, which accounts for an initial rise in throughput at small numbers of processors.)

We can again separate the effects of failure from those of synchronization by having each processor do its Compare-And-Swap to a different memory location. In this case, shown in Figure 7, throughput does not drop off as the number of processors increases. Bus contention due to synchronization is min-

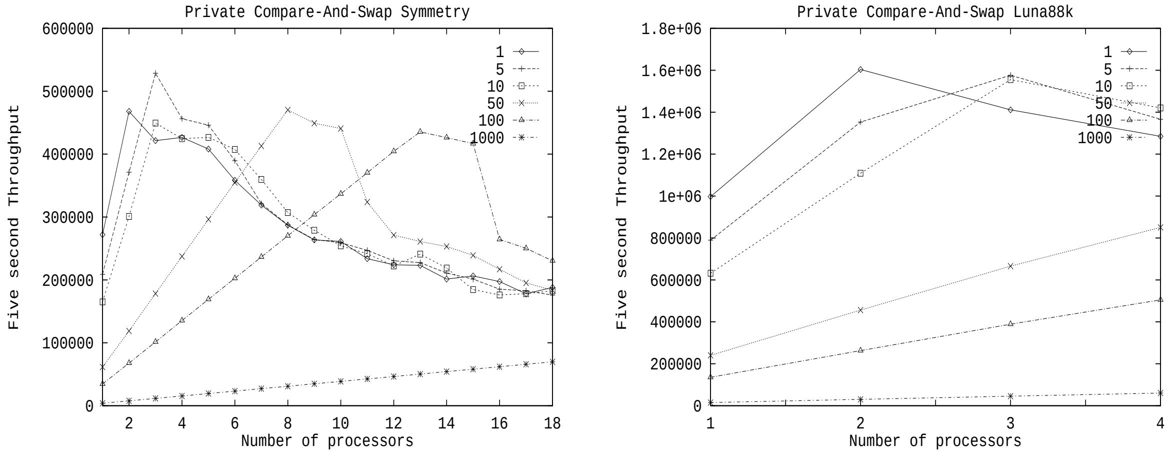

Figure 5: *Compare-And-Swap using spinlocks. Each thread accesses a different location. Throughput drops off less significantly than when each processor manipulates the same location. Contention exists only for the lock, not for the data value.*

imized and every Compare-And-Swap contributes to total throughput. Real concurrent objects, though, require updates to common data, so this experiment is only useful for understanding the effects of queueing and failure.

3.5 Reducing the frequency of wasted synchronization

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. Throughput, the total number of successful accesses to the shared value, is ultimately limited 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 limited, 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 necessarily drops off.

With blocking concurrent objects, that is, those built using traditional locks and critical sections, each synchronization operation is followed by a successful update operation, so synchronization operations are never "wasted." An implementation for Compare-And-Swap should exhibit the same property. 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.

We can build a Compare-And-Swap operation that reduces the cost of failure by changing the definition of Compare-And-Swap so that it becomes *advisory* rather than prescriptive. A failed 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. We can implement the advisory Compare-And-Swap by having a processor poll the old value for equality with the new value while attempting to acquire the lock. A waiting processor can abort its Compare-And-Swap at any time before it acquires the lock 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 synchronized Compare-And-Swap would most likely fail. Successful processors therefore only 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 8. The graphs were generated using Test-And-Test-And-Set spinlocks in which the initial test loop also included a check for equality between the new value and the old value. Compared to the previous techniques, the additional compare substantially reduces the rate at which throughput degrades when processors are added. Moreover, absolute performance in the case of low contention is comparable to the straightforward Compare-And-Swap implementation using simple spinlocks (Figure 4).

3.6 Summary

Synchronization protocols appropriate for non-blocking concurrent objects are inappropriate in terms of performance when used as the basis for a non-blocking concurrent objects. In the presence of contention, simple spinlocks generate excessive bus contention. Queuelocks, which are effective at reducing

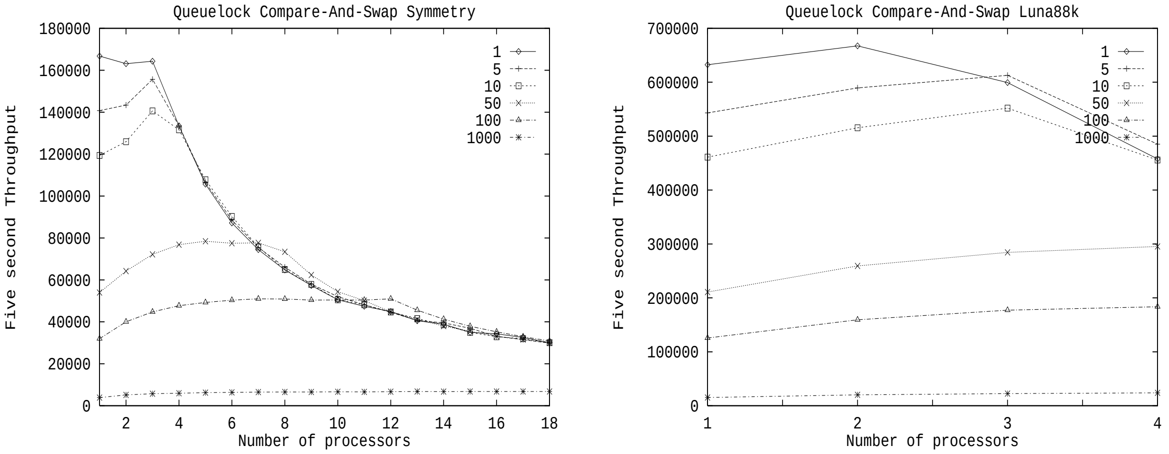

Figure 6: *Compare-And-Swap using queuelocks. Throughput drops off as with the simple spinlock-based implementation.*

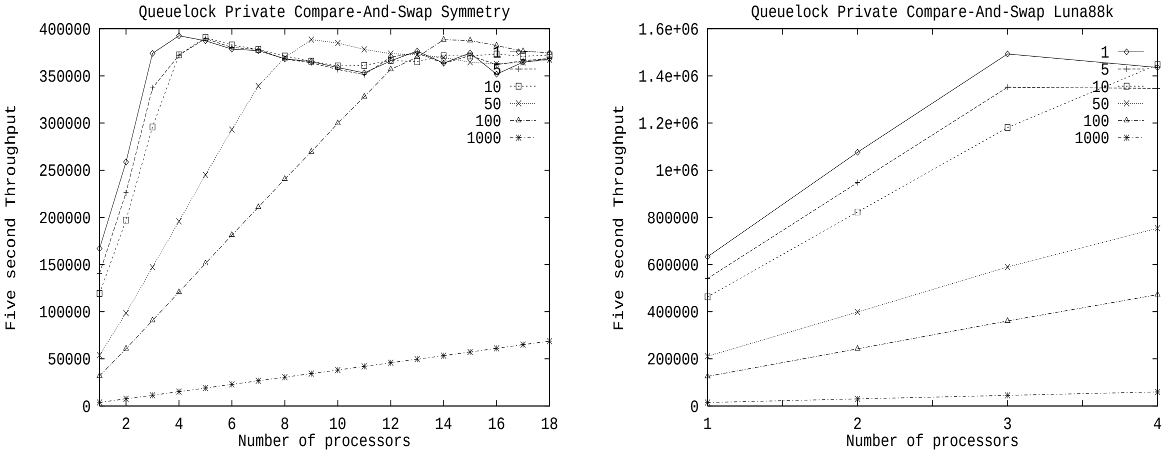

Figure 7: *Compare-And-Swap using queuelocks. Each thread accesses a different location. Throughput does not drop off as processors are added because there is neither lock-contention nor useless lock acquisitions.*

contention with lock-based concurrent objects, are not effective when used with non-blocking 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.

4 Related work

Our use of roll-out is patterned after the optimistic synchronization policies described for Trellis [Moss & Kohler 87], Mach and Taos [Bershad et al. 92], but differs in that it is intended for use on a multiprocessor. As such, performance, as well as progress, is important. Herlihy [Herlihy 90] has shown how to implement non-blocking concurrent objects using the load-linked/store-conditional operation found on architectures such as the MIPS R4000 [Mirapuri et al. 92] and Digital's Alpha [Sites 92]. Herlihy shows that the load-linked/store-conditional operation is more efficient and simpler to use in non-blocking algorithms than Compare-And-Swap (although they are equivalently powerful). As in our work, he examines performance in the presence of contention *and* in the absence of hardware support for the "right" atomic primitive. He does not suggest a strategy for dealing with inopportune preemption when simulating

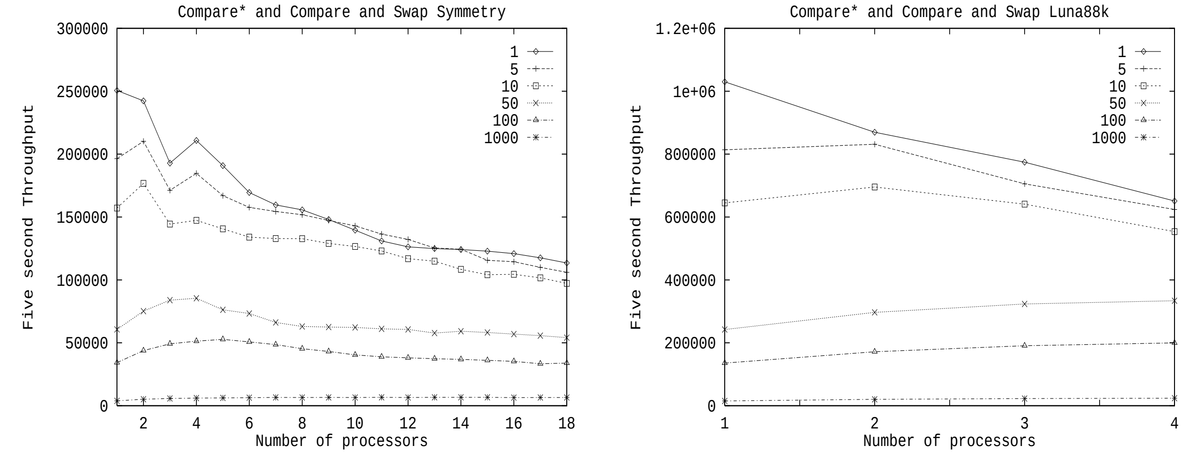

Figure 8: *Compare*-And-Compare-And-Swap. Throughput drops off only gradually or not at all as processors are added. Unnecessary synchronization operations have been eliminated.*

the primitive on architectures that do not provide it. He uses an exponential backoff strategy at the object level, rather than at the synchronization level, to introduce delays during periods of contention and shows that this prevents throughput from dropping off with contention. It is unclear whether the same effect could be obtained by implementing the simulated load-linked/store-conditional with exponential backoff.

Our results do not depend on the fact that Compare-And-Swap is the atomic primitive being simulated. Non-blocking objects based on Herlihy's prescription for load-linked/store-conditional could be built using the techniques described in this paper. The polling aspect of our final policy, Compare*-And-Compare-And-Swap, could be used to implement a store-conditional using synchronization primitives oriented towards blocking synchronization.

While the experiments presented in this paper have been performed on conventional bus-based shared memory multiprocessors, our results and conclusions can be applied to distributed shared memory multiprocessors as well, such as DASH [Lenoski et al. 90] and Alewife [Agarwal et al. 88]. Functionally, these systems are similar to shared memory multiprocessors, in that a parallel program distributed across many processors executes within a single, shared address space. Architecturally, bus-based shared memory machines differ from distributed shared memory machines in that remote memory references (cache misses) generally take substantially fewer processor cycles in a bus-based machine. Since a primary contributor to synchronization overhead in the presence of contention stems from the cost of having to perform operations globally (out of cache), we believe that our results, which reduce the frequency and increase the usefulness of such global operations, apply directly to distributed shared memory multiprocessors.

5 Conclusions

Non-blocking concurrent objects have the potential to become powerful and efficient tools for use in parallel programs. In this paper, we have explored several practical aspects for systems that rely on non-blocking concurrent objects. We have described a simple operating system mechanism with which to build a non-blocking Compare-And-Swap out of blocking primitives such as Test-And-Set. This enables the use of non-blocking objects on a wide range of multiprocessor systems, including both shared memory and distributed shared memory architectures. We have evaluated a set of synchronization policies for implementing the non-blocking mechanisms, and have shown that throughput is vulnerable to contention. Specifically, we have shown that it is the cost of failure with Compare-And-Swap that influences throughput when contention is high. We have shown that it is possible to reduce this cost by relaxing the definition of Compare-And-Swap so that failures can occur without synchronization.

Acknowledgements

Dan Stodolsky helped in evaluating the tradeoffs between various synchronization policies. He, Greg Morrisett, Steve Schwab, and Jeannette Wing provided valuable feedback on this paper's presentation. A conversation with Maurice Herlihy helped me to

convince myself that the ideas in this paper were worth pursuing.

References

[Agarwal et al. 88] Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 280–289, June 1988.

[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. 92] 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. *ACM Transactions on Computer Systems*, 9(1), February 1992.

[Bershad et al. 92] Bershad, B. N., Redell, D., and Ellis, J. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the 5th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

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

[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 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 and Systems*, 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.

[Mirapuri et al. 92] Mirapuri, S., Woodacre, M., and Vasseghi, N. The MIPS R4000 Processor. *IEEE Micro*, 12(4), April 1992.

[Moss & Kohler 87] Moss, J. and Kohler, W. Concurrency Features for the Trellis/Owl Language. In *European Conference on Object-Oriented Programming*, June 1987. Appears in Springer-Verlag's Lecture Notes in Computer Science #276.

[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.

[Sites 92] Sites, R. L. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[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.

[Zahorjan et al. 88] Zahorjan, J., Lazowska, E., and Eager, D. Spinning Versus Blocking in Parallel Systems with Uncertainty. In *Proceedings of the International Seminar on the Performance of Distributed and Parallel Systems*, pages 455–472, December 1988.