Fast Synchronization on Scalable Cache-Coherent Multiprocessors using Hybrid Primitives

Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou
High Performance Information Systems Laboratory
Department of Computer Engineering and Informatics
University of Patras, Rion 26 500, Patras, GREECE

Abstract

This paper presents a new methodology for implementing fast synchronization on scalable cache-coherent multiprocessors, through the use of hybrid primitives*. Hybrid primitives leverage commodity hardware to speed-up the execution of the atomic remote Read-Modify-Write (RMW) instructions employed in synchronization algorithms to resolve contending processors, while exploiting the caches to reduce network traffic during the waiting and release phases of a synchronization primitive. We present a systematic methodology for transforming any synchronization primitive that uses RMW instructions into a hybrid one. We then provide experimental evidence on the effectiveness of using hybrid primitives in the implementation of spin locks, barriers and lock-free queues, in microbenchmarks and parallel applications on a SGI Origin2000.*
keywords: synchronization, shared-memory, cache-coherence, scalable architectures, performance evaluation.

1. Introduction

The proliferation of cache-coherent distributed shared memory multiprocessors and their wide acceptance as viable platforms for parallel and mainstream computing motivate intensive research on understanding the complex hardware/software tradeoffs of these systems and detecting critical performance bottlenecks that may preclude application scalability [20]. Application studies with different benchmark suites and programming models [9, 20] exemplify that the overhead of synchronization remains a dominant bottleneck in parallel programs for cache-coherent shared memory multiprocessors, despite the enormousness of related work that appears in the literature [1, 5, 7, 8, 10, 12, 15, 17]. Synchronization operations account frequently for significant fractions of execution time and may limit the scalability of parallel programs on large processor scales and the ability of large-scale systems to exploit fine-grain parallelism.

During the recent past, researchers investigated software and hardware schemes for implementing fast synchronization primitives. Nevertheless, the effectiveness of previously proposed synchronization mechanisms on scalable cache-coherent multiprocessors is being considerably debated. A recent paper from the authors [18] revealed that several theoretically efficient software synchronization primitives fail to perform well under processor contention, at the moderate scale of 64 processors. Similar results were reported in [11, 12]. Interestingly, the reason behind this behavior does not lie on algorithmic deficiencies. On the contrary, the problem is the memory latency of remote atomic Read-Modify-Write (RMW) instructions that reside on the critical path of software synchronization algorithms. When these instructions are implemented with cacheable variables, they interfere with the directory-based cache coherence protocol and incur undue amounts of network traffic under contention for the synchronization variables.

On the flip side, a related study [11] conjectured that additional hardware support for fast synchronization on scalable cache-coherent systems is not expected to improve significantly application performance and may not justify the high implementation cost. This result is in line with current technology trends for shared memory multiprocessors. System designers tend to avoid the use of specialized synchronization hardware and rely on the atomic instructions embedded in practically all modern microprocessors for implementing software synchronization primitives.

This paper presents *hybrid synchronization primitives*, a new methodology for implementing fast synchronization on scalable cache-coherent shared memory multiprocessors. Our approach lies between hardware and software schemes for fast synchronization and is designed as a generic technique for accelerating any software synchronization primitive that uses atomic RMW instructions. Hybrid primitives leverage commodity hardware to execute the atomic RMW

instructions employed in a synchronization algorithm with a single round-trip network message to the memory module in which the shared synchronization variable is stored, without involving the cache coherence protocol. This sort of hardware can be seamlessly incorporated in scalable cache-coherent systems, without requiring modifications to the microprocessor or the cache coherence protocol. Hybrid primitives exploit the uncached RMW instructions to reduce the latency of the arbitration phase of a synchronization algorithm, during which many processors attempt to atomically modify a shared variable simultaneously. At the same time, hybrid primitives exploit the caches to reduce the number of remote memory accesses and the associated network traffic during the waiting and release phases of a synchronization algorithm.

We analyze the hardware and software requirements for implementing hybrid primitives and present a systematic methodology for transforming any software synchronization primitive into a hybrid one. As case studies, we implemented several hybrid primitives that span a broad spectrum of synchronization algorithms on a 64-processor SGI Origin2000. Using microbenchmarks and three parallel applications, we compare hybrid primitives against implementations of the same primitives that use cacheable synchronization flags and load-linked/store conditional to implement RMW instructions, as well as implementations that use only uncached instructions. Our experimental evidence indicates that hybrid primitives outperform consistently and in many cases significantly their non-hybrid counterparts.

The remainder of this paper is organized as follows: Section 2 provides background and discusses the intuition behind hybrid primitives. Section 3 analyzes the hardware and software requirements for implementing hybrid primitives and presents a systematic methodology for transforming any synchronization primitive into a hybrid one. Section 4 outlines our experimental methodology. Sections 5 and 6 present results with microbenchmarks and applications respectively. Section 7 discusses related work and Section 8 summarizes the paper.

2. Background

We are concerned with the three most frequently used synchronization primitives on shared memory multiprocessors, namely locks, barriers, and lock-free data structures [3]. Locks and barriers have been extensively studied in the past, although thorough evaluations of related algorithms on cache-coherent distributed shared memory multiprocessors had not appeared in the literature until recently [11, 18]. Lock-free synchronization has also attracted considerable attention due to its competitive performance compared to lock-based synchronization and its robustness as a synchronization discipline in multiprogrammed shared memory multiprocessors [2, 14, 17, 18, 21].

Synchronization primitives on shared memory multiprocessors can be analyzed effectively through time decomposition of synchronization periods [6]. A generic synchronization primitive can be decomposed into at most four distinct time intervals, the *acquire*, the *waiting*, the *compute* and the *release* interval. Locks represent the most general case, in which a processor that attempts to atomically access a critical section experiences the latency of all four time intervals on the critical path, with the compute interval corresponding to the critical section itself. Analogous rationale can be developed for barriers, although barriers lack in principle a distinct compute period. In the case of lock-free synchronization [7], although accurate time decomposition depends on the semantics and the type of accesses to the data structure on which a lock-free algorithm operates, update operations are conceptually analogous to acquires.

The essence of a scalable[1] synchronization primitive is to shorten the critical path length of the acquire and the release interval, without implicitly or explicitly dilating neither the compute interval, nor any other useful computation that may be executed concurrently with the synchronization primitive. Such a task is not always trivial, given the complex hardware/software interactions that take place during a synchronization period.

The common thread that connects software synchronization primitives for shared memory multiprocessors is that the acquire is implemented with one or more atomic *Read-Modify-Write (RMW)* instructions on a shared variable, which serves as an arbitration flag for contending processors. Typical atomic RMW instructions are test&set, fetch&add and compare&swap. On a shared-memory multiprocessor with physically distributed memory modules, when multiple processors contend for an arbitration variable, the memory module in which the variable is stored becomes a *hot spot*. If many processors issue RMW instructions to the same module simultaneously, both contention at the memory module and saturation of the interconnection network become critical performance bottlenecks. Network congestion increases the effective latency of memory references and may dilate any useful computation executed by the processors during a synchronization period.

Research on scalable synchronization for shared memory multiprocessors has concentrated on devising algorithms that minimize the number of remote references to synchronization flags during the acquire, waiting and release intervals [1, 5, 6, 8, 10, 15]. These algorithms have proven to be efficient on large-scale shared memory systems without hardware cache coherence. However, the same algorithms implicitly assume that a remote memory access

[1] We define as scalability of a synchronization primitive the ability of the primitive to exhibit nearly constant latency as the number of processors that synchronize increases linearly [15].

costs as much as the round-trip cost of a single network message. This assumption is not valid on scalable shared memory multiprocessors with directory-based cache coherence. If a synchronization flag is cacheable, every update to this flag is followed by invalidations to the caches of processors that share the flag. Each processor that accesses the updated flag experiences a coherence miss and has to fetch the updated copy from a remote memory module. Depending on the cache coherence protocol and the synchronization primitive, a remote RMW reference to a synchronization flag may require a minimum of 5 to 12 non-overlapped network messages [10, 12]. Under heavy contention, the number of invalidations to synchronization flags increases at the square of the number of contending processors. On the other hand, cache coherence has a positive effect on synchronization primitives that use shared flags at the waiting and release intervals. When processors read the flag for the first time, they automatically load the flag in their caches and poll the flag from there, without issuing network messages until the flag is updated by another processor.

Hybrid synchronization primitives use hardware support for executing uncached atomic RMW instructions directly at the memory modules with a single network message. At-memory RMW instructions reduce drastically the coherence-induced overhead of synchronization primitives during the acquire interval. However, using uncached instructions during the waiting interval and at the release is a poor decision, since the induced network traffic may limit the effective bandwidth and degrade performance. Hybrid primitives use therefore cacheable variables for the waiting and the release phases of a synchronization primitive. The use of both uncached and cached memory instructions in the same synchronization primitive motivates the name hybrid primitives.

Figure 1 gives a quantitative view of the intuition behind hybrid primitives. Chart (a) illustrates the execution time of a microbenchmark which performs 10000 atomic fetch&add's to a shared counter, executed on a 64-processor SGI Origin2000. The chart shows that the latency of a regular cache-coherent fetch&add can be as much as 5 times the latency of an uncached fetch&add, executed directly at the memory modules. Chart (b) illustrates results from a microbenchmark where processors access repeatedly a critical section in a round-robin manner. Access to the critical section is controlled with a global ticket number similarly to the bakery algorithm [15]. Each processor waits outside the critical section until its private ticket number becomes equal to the global ticket number. The figure illustrates that it is preferable to allocate the waiting flag in a cacheable memory location. Having processors busy-waiting on an uncacheable location dilates the critical section and the release interval, resulting to a slowdown of 46% on 64 processors.

3. Hybrid Synchronization Primitives

This section discusses the software and hardware requirements of hybrid primitives and presents a concrete methodology for their implementation.

3.1. Hardware and Software Requirements

Hybrid primitives require hardware support for implementing atomic RMW instructions directly at the memory modules of a cache-coherent multiprocessor and software support for mapping regions of the memory modules to the virtual address space of user programs as uncacheable. An important implementation consideration is the granularity at which uncacheable memory regions can be allocated. State-of-the-art microprocessors support the allocation of uncacheable memory regions at the granularity of a page. Therefore, uncacheable variables must be situated in different pages than cacheable variables. It would be desirable to allocate uncacheable regions at a finer granularity and be able to effectively collocate cacheable and uncacheable variables on the same page. Such an implementation would reduce memory consumption and increase flexibility in the implementation of hybrid primitives.

Processors can access and modify variables allocated in uncacheable regions by issuing load and store instructions which are executed with a single round-trip message. It is not necessary to modify the microprocessor's instruction set architecture in order to accommodate uncached read, write, or RMW instructions. The uncached instructions can be encoded in the address of regular loads and stores. The high-order address bits of the instruction can map to a designated uncacheable memory region, while the offset of the instruction can encode the type of operation to be applied to the specified memory location. The memory controller can then translate the load/store instruction into the appropriate operation at the specified location in physical memory. This solution is adequate to implement atomic RMW instructions with at most two arguments. For more powerful primitives that require three arguments, e.g. compare&swap, it is possible to add one or more external registers (similar of the E-registers of the Cray T3E [19]) and encode the address of the register used for the comparison in the unused bits of the argument of the uncached instruction.

At the software level, uncacheable variables and atomic RMW instructions on these variables can be explicitly tagged, in order to be translated into the associated load/store instructions. A more convenient solution is to have the compiler directly recognize uncached variables by their tags at their declaration points and transparently translate accesses to these variables into the appropriate uncached memory instructions.

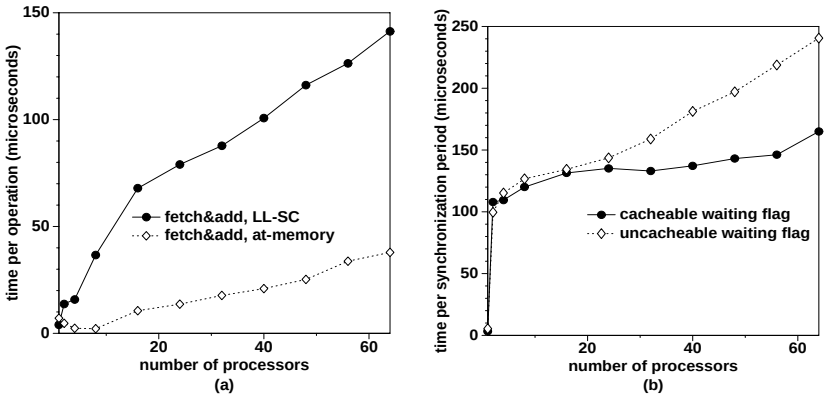

Figure 1. A quantitative view of hybrid primitives.

3.2. Methodology

Our generic methodology for implementing hybrid primitives is to logically decouple the acquire from the wait and release intervals of a synchronization period by using two separate synchronization variables, one uncacheable variable for the acquire and one or more cacheable variables for the wait and the release. This type of variable splitting is already inherent in several synchronization algorithms, such as queuing locks and sense-reversing barriers. In these cases, transforming the primitive into hybrid is as simple as allocating the two synchronization flags in two separate memory regions, one of which is uncacheable, and tagging the uncacheable variables and the RMW instructions on these variables.

There are two cases in which embedding "hybridity,, in a synchronization primitive is not straightforward. The first case is when a single flag is used for both the acquire and the wait/release operations. The second case is when the applied synchronization algorithm uses a pointer-based data structure, such as a linked list, which may be accessed concurrently by multiple processors.

The solution in the first case is to allocate the acquire flag in an uncacheable region and use an additional cacheable flag for the release. The acquire can be implemented with the accelerated uncached RMW instructions. At the release, the uncacheable flag is reset and its new value is loaded directly into the cacheable waiting flag with an uncached MWR (modify-write-then-read) instruction. This comes at the expense of an additional message for resetting the uncacheable flag, with the prospect that this overhead will be offset by the gain from performing the RMW instructions faster.

The case of synchronization algorithms that use pointer-based data structures is to some extent more complicated. Pointer-based data structures are implicitly represented by a set of access points to the data structure (e.g. the head of a queue or the root of a tree) and a number of linkage pointers within the nodes of the data structure, which maintain the data structure's topology. Synchronization algorithms need remote RMW instructions to update the access points to the data structure, while updates of the linkage pointers require remote read, write and in some cases RMW instructions. The problem with pointer-based data structures is that the nodes of the data structure contain processor-private information, such as local waiting flags and data which is accessed by dereferencing the linkage pointers. In principle, the cacheability of this data should be preserved in a hybrid implementation.

Ideally, a hybrid implementation would allocate the access points and the linkage pointers at uncacheable locations, in order to minimize the cost of remote memory accesses, while maintaining the other fields in cacheable locations. Such an implementation however has two implicit requirements: first, that the system can collocate cacheable and uncacheable variables in the same memory page; and second that compiler support is provided so that uncacheable variables are accessed and dereferenced transparently as regular pointers. Although the second requirement could be satisfied relatively easily, the first requirement may be hard to satisfy, as it depends on the granularity at which the system allocates uncacheable memory regions. If the granularity of allocation is equal to a virtual memory page, collocation of cacheable and uncacheable variables is not feasible. In that case, accessing an uncached linkage pointer would require an extra level of indirection, i.e. one remote access to obtain the linkage pointer from the uncached location and an additional cached remote access after dereferencing the pointer. This solution would clearly add overhead on the critical path of a synchronization primitive.

If hardware support for collocating cacheable and uncacheable variables is not available, the best solution for a hybrid primitive is to maintain the linkage pointers at

```
typedef struct {
  padded unsigned long c_lock;
  uncacheable unsigned long *u_lock;
} lock_t;

void lock (lock_t *l) {
  int i;

  while (l->c_lock==1)
   exp_backoff();

  while (uncached_test_and_set (l->u_lock)==1) {
    while (l->c_lock==1)
      exp_backoff();
  }
}

void unlock (lock_t *l) {

  l->c_lock=uncached_store_and_fetch (l->ulock, 1);
}
```

Figure 2. The hybrid implementation of the test&set lock.

Locks		
Algorithm	**Acquire**	**Release**
test&test&set, LL-SC	cached	cached
test&test&set, at-memory	uncached	uncached
test&test&set, hybrid	uncached	cached
array queue lock, LL-SC	cached	cached
array queue lock, hybrid	uncached	cached
ticket lock, LL-SC	cached	cached
ticket lock, at-memory	uncached	uncached
ticket lock, hybrid	uncached	cached
Barriers		
Algorithm	**Acquire**	**Release**
counter, LL-SC	cached	cached
counter, at-memory	uncached	uncached
counter, hybrid	uncached	cached
MCS tree barrier	cached	cached
Lock-Free Queues		
Algorithm	**Access Points**	**Data Nodes**
MS non-blocking queue	cached	cached
fetch&add, LL-SC	cached	cached
fetch&add, at-memory	uncached	uncached
fetch&add, hybrid	uncached	cached

Table 1. Synchronization primitives evaluated on the SGI Origin2000.

cacheable locations and allocate as uncacheable only the variables that hold the access points to the data structure. Modifications to the access points can then be implemented with the fast atomic RMW instructions at the memory modules. The RMW instructions return linkage pointer values which can be directly stored in cacheable variables and manipulated regularly by the compiler. This solution comes at the expense of increased memory consumption for the synchronization data structures.

Figure 2 illustrates as an example the hybrid implementation of the test&set lock.

4 Evaluation Framework

We evaluate synchronization primitives using both microbenchmarks and real applications. Our microbenchmarks execute repeatedly a synchronize() function, which may be any of a lock/unlock pair enclosing a critical section, a barrier, or an access to a lock-free queue. Inside critical sections and during the delay intervals, each processor modifies a shared vector, thus requiring remote memory accesses to obtain updated copies of the shared cache lines. The choice of accessing remote memory in the delay intervals and during critical sections is made to evaluate the impact of the synchronization primitive on any useful computation executed concurrently with synchronization. In the microbenchmarks, we vary the critical parameters of a synchronization period, that is, the length of the critical section in the case of locks, the interval between synchronization periods and the load imbalance within this interval.

Our metric for evaluating synchronization algorithms is the average execution time of a synchronization period i.e. an acquire-wait-compute-release sequence, including the delay between successive synchronization periods. This metric provides an accurate view of the performance of a synchronization primitive, because it accounts for the overhead of the actual synchronization primitive and the interferences between the synchronization primitive and the computation performed by processors during delay intervals and critical sections. In the case of parallel applications, we use absolute speedups as the performance metric.

We used a 64-processor SGI Origin2000 as our evaluation testbed. The Origin2000 is suitable for our purposes, since it provides specialized hardware and software support for executing atomic read, write, and RMW instructions directly at uncacheable memory locations [13]. The implementation of these instructions, called *fetchops*, follows roughly the scheme outlined in Section 3.1. The Origin2000 fetchop hardware implements atomic uncached reads, writes, and/or instructions, fetch&increment, fetch&decrement, and swap with zero. An obvious drawback is the absence of a powerful atomic primitive such as an atomic register-memory swap, or a compare&swap. This limitation precludes the implementation of non-blocking lock-free algorithms and list-based locks [7]. Furthermore, the system does not allow the collocation of cacheable and uncacheable variables on the same page. On the other hand, the Origin2000 nodes are equipped with a 1-entry cache for storing the contents of the most recently accessed fetchop variable. This cache can accelerate operations such as polling of a waiting flag.

Table 1 summarizes the synchronization primitives that we evaluate in this paper. Whenever possible, we provide three implementations for each primitive. The first

implementation uses regular cache-coherent synchronization variables and atomic RMW instructions implemented with the load linked/store conditional (LL-SC) instruction of the MIPS R10000 microprocessors of the Origin2000. The second implementation uses solely uncached synchronization instructions. This implementation is tuned specifically for the Origin2000 by experimentally adjusting the backoff parameters, as described in an earlier paper [18]. The third implementation is the hybrid implementation that follows the methodology presented in Section 3. Algorithms that use compare&swap, as well as algorithms that need alignment of uncached variables in the Origin2000 memory modules could not be implemented due to the Origin2000 hardware limitations. The latter was the reason for which an array queue lock could not be implemented using only uncached instructions. The implementations of hybrid primitives are available via anonymous ftp at `venus.hpclab.ceid.upatras.gr`.

5 Results with Microbenchmarks

This section presents our results with microbenchmarks. The following charts report the mean execution time of a synchronization period, averaged over 10 experiments with 1000 synchronization periods executed in each experiment. An artificial load imbalance of $\pm 10\%$ was added in the delay intervals between synchronization periods. The lengths of non-empty delay periods and critical sections reported in the results are for the single-processor case.

5.1 Results

Figure 3 illustrates the results from executions of the microbenchmarks for locks, with empty and non-empty critical sections and delay intervals. Comparing the net overhead of the lock primitives, we witness that the hybrid implementations are uniformly faster than the implementations with LL-SC. The hybrid test&set lock is 2.2 times faster than the LL-SC lock. The hybrid ticket lock is also 2.2 times faster than the LL-SC ticket lock. The hybrid queue lock is 2.6 times faster than the LL-SC queue lock. Similar results are obtained when comparing the hybrid ticket lock against the at-memory ticket lock. The at-memory test&set lock however, compares more favorably to the hybrid implementation, being only 15% slower. The competitive performance of the at-memory test&set lock is attributed to the effectiveness of the fetchop cache of the Origin2000 nodes, which holds the lock while processors busy-wait outside the critical section.

For realistic microbenchmark parameters with non-empty critical sections and delay intervals, hybrid locks still provide sizeable performance improvements compared to LL-SC and at-memory locks. The hybrid test&set lock is 90% faster than the LL-SC test&set lock and 31% faster than the at-memory test&set lock. The hybrid ticket lock is 16% faster than the LL-SC ticket lock and 84% faster than the at-memory ticket lock. The hybrid queue lock is 14% faster than the LL-SC queue lock. The good performance of the at-memory test&set lock is again attributed to the fetchop cache. The good performance of the LL-SC implementation of the ticket lock is attributed to the effectiveness of the backoff strategy used in the experiment. On the other hand, the at-memory ticket lock suffers from the inability of the fetchop cache to hold both the acquire and the release flags. The hybrid queue lock and the LL-SC queue lock differ only in the implementation of the fetch&add instruction that acquires the lock, therefore the improvement is modest. Linked-list implementations of queuing locks require at least two atomic RMW instructions and would benefit more from the hybrid implementation. We implemented two list-based queuing locks, the MCS queuing lock [15] and Fu's circular list-based lock [5], using LL-SC to implement the required RMW instructions and observed that their performance was 20%-30% worse than the performance of the array-based queuing lock due to the high execution cost of compare&swap on the Origin2000[2].

Figure 4 shows the results from executions of the microbenchmarks for barriers with two delay period lengths. With an empty delay period, the hybrid counter barrier is 2.4 times faster that the LL-SC counter barrier and 36% faster than the MCS tree barrier. The hybrid counter barrier is only 20% faster than the at-memory barrier, because of the effectiveness of the fetchop cache of the Origin2000 nodes, which satisfies most reads to the waiting flag of the barrier.

Figure 5 illustrates the results from executions of the microbenchmarks for lock-free queues with empty and non-empty delay periods. In all cases the hybrid fetch&add queue outperforms the at-memory and the LL-SC implementations, by at least 33% and at most 170%. The striking difference of lock-free queues as opposed to locks and barriers is that they have no distinct waiting phase in the critical path. Instead, the latency of the enqueue/dequeue operations is proportional to the number of RMW instructions issued to maintain the queue consistent. Hybrid implementations accelerate all these RMW instructions and provide solid performance improvements. On the other hand, using uncacheable variables for the queue nodes leads to inferior performance, due to a hot spot incurred from uncached accesses to the node that holds the queue array. As a result, the performance of the at-memory implementation is in most cases inferior to the performance of the LL-SC

[2] On a directory-based cache-coherent system, the array-based queuing lock and the MCS lock have three remote accesses on the critical path, however the MCS lock uses compare&swap which is significantly slower than fetch&add on the Origin2000. Fu's circular list lock has two expensive remote RMW instructions on the critical path, namely a fetch&store and a swap&compare.

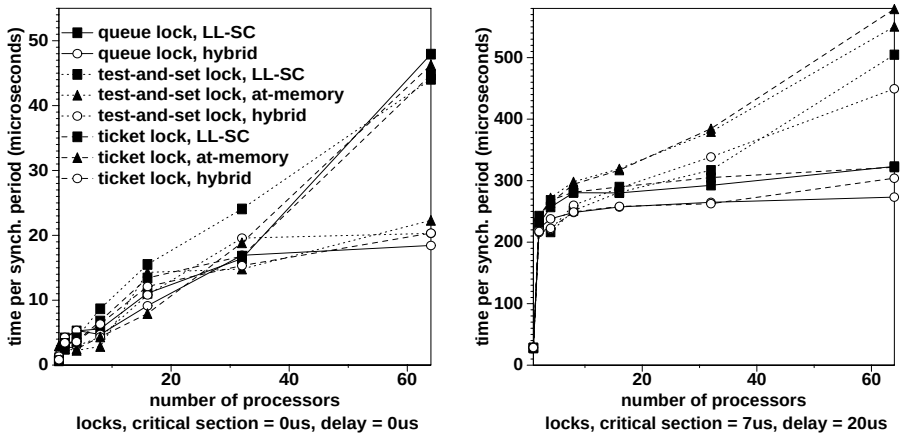

Figure 3. Microbenchmark results for locks.

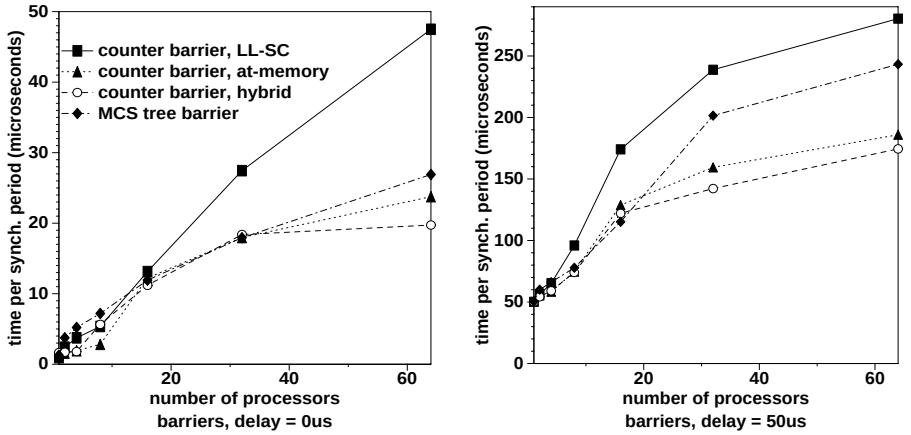

Figure 4. Microbenchmark results for barriers.

implementation. Michael and Scott's queue underperforms the fetch&add queue due to the existence of two slow compare&swap's and several remote poll instructions on the critical path of enqueue/dequeue operations. This additional cost is paid in order to preserve the non-blocking property of the queue.

To summarize, we conclude that the results for hybrid synchronization primitives are encouraging, since the primitives provide sizeable and in many cases large performance improvements for all the algorithms with which we experimented on the Origin200. Our results can be characterized as conservative, if one takes into account the specific features of the hardware platform on which we experimented. The Origin2000 has a very aggressive memory and communication architecture. The remote-to-local memory access latency ratio of the system is no more than 3 to 1 for configurations up to 128 processors. As a result, the potential for improving synchronization primitives by speeding up remote RMW instructions on the Origin2000 is not large. We would expect larger improvements on other commercial systems, in which the remote to local memory access latency ratios are significantly higher compared to that of the Origin2000.

6 Evaluation with Applications

Our main consideration in the evaluation of hybrid synchronization primitives with parallel applications is to demonstrate the performance of the primitives under more realistic conditions compared to those established with our microbenchmarks and examine whether hybrid primitives can offer appreciable performance improvements for applications with fine-grain parallelism. As soon as fine-grain parallelism does exist in numerous applications and synchronization overhead is often the obstacle towards exploiting this type of parallelism, evaluating the scalability of complete application benchmarks with different synchronization primitives is worthwhile. However, we do not claim that using a fast synchronization primitive is the primary means to improve the scalability of parallel programs

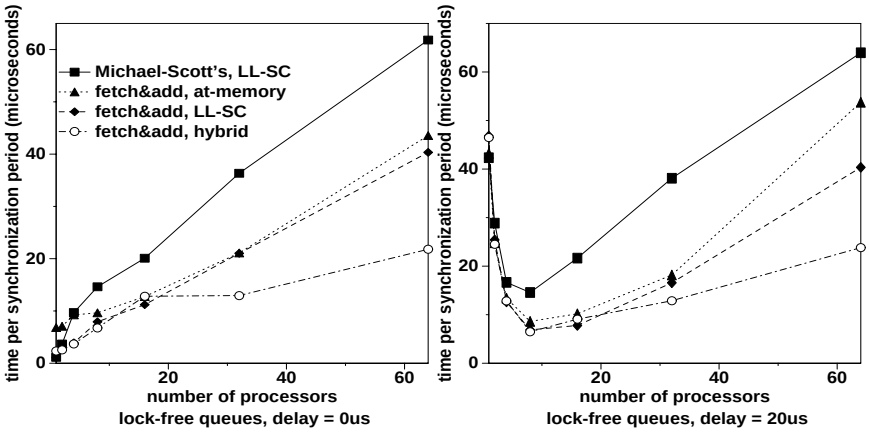

Figure 5. Microbenchmark results for lock-free queues.

in large-scale shared memory multiprocessors. On the contrary, we believe that scaling shared memory programs to hundreds of processors requires algorithmic modifications to minimize the frequency or the universality of synchronization [4, 9], before considering the use of better synchronization primitives.

We used three applications from the SPLASH-2 benchmark suite [22], Radiosity, and Barnes for the evaluation of locks and LU for the evaluation of barriers. All three applications have frequent calls to synchronization primitives and fine-grain synchronization patterns. The problem sizes were selected to ensure satisfactory scalability of the applications on the system on which we experimented, for at least one of the synchronization primitives in each case. We selected the small problem size for Radiosity, a 16K-bodies problem size for Barnes and a 128×128 problem size for LU.

Figure 6 illustrates the speedups of the three benchmarks when executed with different lock and barrier synchronization primitives on the SGI Origin2000. In all cases, a hybrid primitive attains the best speedup and at least one of the hybrid primitives sustains acceptable scalability up to 64 processors. The performance difference between the best hybrid lock and the best non-hybrid lock in Radiosity is 15%. The rest of the non-hybrid primitives force the application to run two to four times slower on 64 processors, compared to the hybrid primitives. The improvements from using hybrid primitives in Barnes are modest, since this application suffers less from contention during synchronization periods. The hybrid barrier in LU is the only primitive that achieves scalability beyond 16 processors.

7 Related work

A classic paper by Mellor-Crummey and Scott [15] initiated a series of studies on fast software synchronization algorithms for shared memory multiprocessors. This paper concluded that effective exploitation of the memory hierarchy in software is sufficient for implementing scalable synchronization algorithms, without the need for expensive specialized synchronization hardware. Motivated from this work, several researchers have investigated software algorithms that reduce the number of remote memory accesses needed to implement synchronization primitives [5, 8]. These algorithms were in principle implemented and evaluated on real or simulated shared memory multiprocessors without coherent caches. This is a key difference to our work, which addresses the problem of achieving fast synchronization on scalable cache-coherent systems. Furthermore, our approach is orthogonal to previous proposals for improving synchronization algorithms in software. Hybrid primitives constitute a generic methodology to accelerate any synchronization primitive that uses RMW instructions, without modifying the employed synchronization algorithm.

Our study corroborates previous results from an earlier paper of Michael and Scott [16] which investigated alternative implementations of atomic RMW instructions on scalable cache-coherent multiprocessors. This study concluded that implementing RMW instructions at the memory nodes can be beneficial for heavily contended synchronization variables. Our work extends this result by effectively embedding uncached RMW instructions in software synchronization algorithms to improve their scalability.

Kuo et.al. [12] proposed the use of message passing for the implementation of locks in shared memory multiprocessors. Their work, similarly to ours, was motivated by the problem of the latency of remote memory accesses of lock operations on scalable cache-coherent systems. Message-passing locks use a kernel-level lock manager which mediates lock requests and transfers lock privileges with messages. Their implementation builds on the assumption

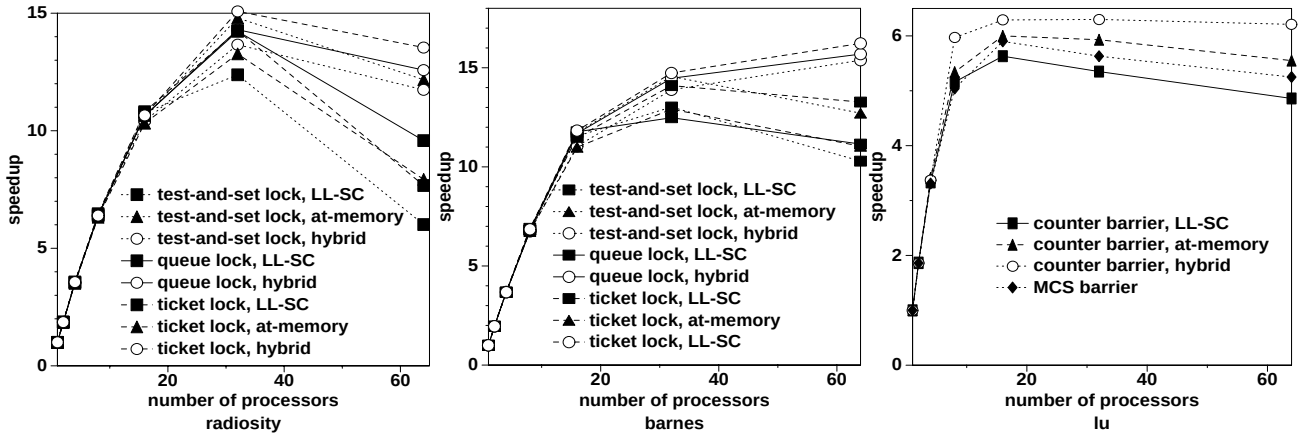

Figure 6. Results from application benchmarks.

that message-passing can be very fast on modern shared-memory multiprocessors. Message-passing locks can provide large performance improvements for systems with slow interconnection networks, however they tend to underperform other software locks on systems with fast interconnection networks [12]. Another drawback of message passing locks is that they require modifications to the operating system in order to be implemented efficiently. Hybrid primitives are simple to implement and have exhibited solid performance improvements on a system with very fast and sophisticated interconnection architecture.

Building specialized hardware for fast synchronization is an active topic of research, especially in the area of interconnection networks. The major concern about hardware implementations of synchronization primitives is their high implementation cost and the trend in state-of-the-art shared memory multiprocessors is to avoid using dedicated synchronization hardware, as soon as scalable synchronization primitives can be implemented in software [19]. The most viable proposal for hardware synchronization appears to be the Queue on Lock Bit (QOLB), originally proposed in [6]. QOLB is the hardware incarnation of a list-based queuing lock with local spinning. However, the hardware implementation enables a processor that holds a lock to transfer it to another processor with a single cache-to-cache transfer. QOLB requires the absolute minimum number of remote messages for a lock acquire/release pair. As with other hardware implementations of synchronization primitives, QOLB's main drawback is its implementation cost and complexity. QOLB requires significant modifications to the processor's instruction set architecture, the cache controller and the cache-coherence protocol in order to fully exploit its benefits [10]. A simulation study of a hardware implementation of QOLB [10] reported that hardware QOLB is able to provide up to 2.7-fold improvements in application execution times. We witnessed analogous improvements with hybrid primitives on a real system, although we bear in mind that the hardware platform and the coherence protocol used in our study differ significantly from the experimental setting of the QOLB study [10].

8 Conclusions

This paper presented a new methodology for implementing fast synchronization on scalable cache-coherent multiprocessors. Hybrid primitives leverage commodity hardware to reduce the latency of the expensive remote RMW instructions and accelerate the execution of any software synchronization algorithm that employs this type of instructions. We presented a systematic methodology for transforming software synchronization primitives into hybrid ones. As case studies, we implemented several hybrid primitives on a 64-processor SGI Origin2000 and evaluated them against well-tuned alternative implementations with microbenchmarks and parallel applications.

Besides the results, this paper takes a new approach for synchronization on shared memory multiprocessors, which lies between aggressive hardware and software mechanisms. In line with several researchers, we argue that using expensive specialized synchronization hardware is not necessary for achieving fast synchronization on large-scale systems. However, directory-based cache coherence on shared memory multiprocessors poses the obstacle of remote memory latency, which can not be easily hidden or tolerated in synchronization operations. We have shown that using commodity hardware to bypass cache coherence only for the purposes of synchronization can alleviate the latency bottleneck effectively.

Acknowledgments

This work was partially supported by the E.C. through Framework IV (ESPRIT Programme, Project No. 21907,

NANOS). The authors would like to thank the European Center for Parallelism in Barcelona (CEPBA), for providing the resources to conduct the experiments presented in this paper.

References

[1] T. Anderson. *The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors*. IEEE Trans. on Parallel and Distributed Systems, 1(1), pp. 6–16, Jan. 1990.

[2] N. Arora, R. Blumofe and C. Greg-Plaxton. *Thread Scheduling for Multiprogrammed Multiprocessors*. Proc. of the 10th ACM Symp. on Parallel Algorithms and Architectures, pp. 119–129, Puerto Vallarta (Mexico), Jun. 1998.

[3] D. Culler, J. P. Singh and A. Gupta. *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufmann, Aug. 1998.

[4] P. Diniz and M. Rinard. *Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs*. Journal of Parallel and Distributed Computing, 49(2), pp. 218–244, 1998.

[5] S. Fu and N. Tzeng. *A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors*. IEEE Trans. on Parallel and Distributed Systems, 8(6), pp. 628–639, Jun. 1997.

[6] J. Goodman, M. Vernon and P. Woest. *Efficient Synchronization Primitives for Large-Scale Cache-Coherent Shared-Memory Multiprocessors*. Proc. of the 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 64–75, Boston (USA), Apr. 1989.

[7] M. Herlihy. *Wait-free Synchronization*. ACM Trans. on Programming Languages and Systems, 13(1), pp. 124–149, Jan. 1991.

[8] T. Huang. *Fast and Fair Mutual Exclusion for Shared Memory Systems*. Proc. of the 19th IEEE Int. Conf. on Distributed Computing Systems, pp. 224–231, Austin (USA), Jun. 1999.

[9] D. Jiang and J. P. Singh. *Scaling Application Performance on a Cache-Coherent Multiprocessor*. Proc. of the 26th Int. Symp. on Computer Architecture, pp. 305–316, Atlanta (USA), May 1999.

[10] A. Kägi and J. Goodman. *Efficient Synchronization: Let Them Eat QOLB*. Proc. of the 24th Int. Symp. on Computer Architecture, pp. 171–181, Denver (USA), Jun. 1997.

[11] S. Kumar, D. Jiang, R. Chandra and J. P. Singh. *Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance.* Proc. of the 1999 ACM SIGMETRICS Conf., pp. 23–34, Atlanta (USA), May 1999.

[12] C. Kuo, J .Carter and R. Kuramkote. *MP-LOCKs: Replacing H/W Synchronization Primitives with Message Passing*. Proc. of the 5th Int. Symp. on High Performance Computer Architecture, pp. 284–288, Orlando (USA), Jan. 1999.

[13] J. Laudon and D. Lenoski. *The SGI Origin2000: A ccNUMA Highly Scalable Server*. Proc. of the 24th Int. Symp. on Computer Architecture, pp. 241–251, Denver (USA), Jun. 1997.

[14] S. Lumetta and D. Culler. *Managing Concurrent Access for Shared Memory Active Messages*. Proc. of the 12th IEEE Int. Parallel Processing Symp., pp. 272–278, Orlando (USA), Apr. 1998.

[15] J. Mellor-Crummey and M. Scott. *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*. ACM Trans. on Computer Systems, 9(1), pp. 21–65, Feb. 1991.

[16] M. Michael and M. Scott. *Implementation of Atomic Primitives on Distributed Shared Memory Multiprocessors*. Proc. First Int. Symp. on High Performance Computer Architecture, pp. 222–231, Jan. 1995.

[17] M. Michael and M. Scott. *Non-blocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared Memory Multiprocessors*. Journal of Parallel and Distributed Computing, 54(2), pp. 162–182, Feb. 1998.

[18] D. Nikolopoulos and T. Papatheodorou. *A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin2000*. Proc. of the 13th ACM Int. Conf. on Supercomputing, pp. 319–328, Rhodes (Greece), Jun. 1999.

[19] S. Scott. *Synchronization and Communication in the T3E Multiprocessor*. Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 26–36, Cambridge (USA), Oct. 1996.

[20] Y. Solihin, V. Lahm and J. Torellas. *Scal-Tool: Pinpointing and Quantifying Scalability Bottelenecks in DSM Multiprocessors*. Proc. of Supercomputing'99, Portland(USA), Nov. 1999.

[21] H. Tang, K. Shen and T. Yang. *Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines*. Proc. of the 7th ACM Symp. on Principles and Practices of Parallel Programming, pp. 107–118, Atlanta(USA), May 1999.

[22] S. C. Woo, M. Ohara, E. Torrie. J. P. Singh and A. Gupta. *The Splash-2 Benchmarks: Characterization and Methodological Considerations*. Proc. of the 22nd Int. Symp. on Computer Architecture, pp. 24–36, St. Margherita Ligure (Italy), Jun. 1995.