The Architectural and Operating System Implications on the Performance of Synchronization on ccNUMA Multiprocessors

Published in:
International Journal of Parallel Programming

The Architectural and Operating System Implications on the Performance of Synchronization on ccNUMA Multiprocessors

Dimitrios S. Nikolopoulos[1,2] and
Theodore S. Papatheodorou[3]

This paper investigates the performance of synchronization algorithms on ccNUMA multiprocessors, from the perspectives of the architecture and the operating system. In contrast with previous related studies that emphasized the relative performance of synchronization algorithms, this paper takes a new approach by analyzing the sources of synchronization latency on ccNUMA architectures and how can this latency be reduced by leveraging hardware and software schemes in both dedicated and multiprogrammed execution environments. From the architectural perspective, the paper identifies the implications of directory-based cache coherence on the latency and scalability of synchronization instructions and examines if and how can simple hardware that accelerates these instructions be leveraged to reduce synchronization latency. From the operating system's perspective, the paper evaluates in a unified framework, user-level, kernel-level and hybrid algorithms for implementing scalable synchronization in multiprogrammed execution environments. Along with visiting the aforementioned issues, the paper contributes a new methodology for implementing fast synchronization algorithms on ccNUMA multiprocessors. The

---

[1] This work was carried out while the author was with the High Performance Information Systems Laboratory, University of Patras, Greece.

[2] Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1308 West Main Str., Urbana, Illinois 61801. E-mail: dsn@csrd.uiuc.edu

[3] Department of Computer Engineering and Informatics, University of Patras, GR26500, Patras, Greece. E-mail: tsp@hpclab.ceid.upatras.gr

relevant experiments are conducted on the SGI Origin2000, a popular commercial ccNUMA multiprocessor.

1. INTRODUCTION

The proliferation of ccNUMA multiprocessors as a common platform for parallel and mainstream computing motivates intensive efforts on understanding the architectural implications on the performance of these systems and identifying the critical bottlenecks that limit their scalability. Several evaluations of ccNUMA multiprocessors have exemplified that the overhead of synchronization is frequently the dominant bottleneck towards sustaining high performance.[1, 2] Synchronization overhead has three undesirable effects: It precludes the scalability of parallel programs to large processor scales despite the availability of sufficient algorithmic parallelism; it hinders the ability of the system to exploit fine-grain parallelism; and degrades the performance of the operating system scheduler in terms of throughput and processor utilization, by wasting useful processor cycles in busy-wait loops due to inefficient scheduling of synchronizing threads.

1.1. Motivation

The overhead of synchronization on shared-memory multiprocessors stems from three sources. The first is the latency of synchronization primitives. The second is the overhead of waiting at the synchronization points of a parallel program, due to load imbalance in the parallel computation. The third is the intervention of the operating system's scheduler, when a parallel program with a fine-grain synchronization pattern executes in a multiprogrammed environment. The problem arises when the operating system decides to preempt a thread that executes on the critical path of a synchronization operation.

Although synchronization was the subject of a significant volume of research papers[3–5] in the past, evaluations of synchronization algorithms on commercial ccNUMA multiprocessors had not appeared in the literature until recently.[6, 7] [Note: An overview of the related literature is out of the scope of this paper. The reader can refer to Anderson;[3] Gupta *et al.*;[4] and Mellor-Grummey and Scott[5] for thorough investigations of synchronization algorithms on shared-memory multiprocessors.] Previous related studies were based on execution-driven simulations of cache-

coherent distributed shared-memory multiprocessors with scaled-down versions of popular parallel benchmarks.[8–11]

Synchronization algorithms for shared-memory multiprocessors in general have attracted considerable interest during the last two decades. Several investigators have proposed scalable algorithms for spin locks and barriers.[3, 5, 12–15] These algorithms were evaluated on either small-scale bus-based multiprocessors, or dance-hall distributed shared-memory architectures with no hardware support for cache coherence. Others proposed aggressive hardware implementations of synchronization primitives and optimizations of the cache-coherence protocol to accelerate the execution of critical sections and barriers on ccNUMA architectures.[8, 10, 11, 16] A third proposal was to move the functionality of synchronization primitives inside the operating system and leverage message passing for the implementation of scalable mutual exclusion algorithms.[9] The diversity of the approaches and the conclusions extracted from these works demonstrate that the trade-off's between hardware and software implementations of synchronization primitives on ccNUMA multiprocessors are not clear. [Note: In this paper, the term *scalability* of a synchronization algorithm refers to the ability of the algorithm to execute with nearly constant latency as the number of processors that synchronize increases linearly.] Assessing the relative performance of hardware and software implementations of synchronization algorithms on ccNUMA systems is an important topic of interest for both application and system software developers. It may also serve as a guideline for the design and implementation of ccNUMA architectures in the future. Evaluating software and hardware implementations of synchronization mechanisms in a unified context on an actual commercial ccNUMA system is the primary motivation of this paper.

The relative performance of combinations of *waiting algorithms*,[4] synchronization algorithms and operating system scheduling strategies for multiprogrammed ccNUMA multiprocessors has not been investigated in a unified framework in the past. Moreover, the related issues have not been investigated on actual ccNUMA systems yet. The last relevant evaluation was performed with a simulator of the Stanford DASH multiprocessor.[4] Since then, several researchers studied the interference between scheduling and synchronization, using either of two implicit assumptions. Some assumed that the operating system uses a time-sharing scheduler and attempted to improve the waiting algorithms used at synchronization points. Others assumed that threads always busy-wait at synchronization

---

[4] The term *waiting algorithm* refers to the algorithm used by the threads of a parallel program, whenever the threads reach a synchronization point. The waiting algorithm may also involve scheduling actions on behalf of the threads that synchronize.

points and attempted to improve the scheduling policy of the operating system, in order to reduce the waiting overhead of synchronization. Under the assumption of a time-sharing scheduling strategy, researchers have attempted to embed a degree of adaptability in the waiting algorithms, in order to cope with the undesirable effects of inopportune preemptions of threads by the operating system.[17–19] On the flip side, under the assumption of a busy-wait synchronization algorithm, researchers have proposed scheduling strategies based on gang scheduling, to ensure that the synchronizing threads of parallel programs are always coscheduled.[20, 21] An important question that has not been answered clearly yet is whether a time-sharing scheduler with a scheduler-conscious synchronization algorithm, or a gang scheduler with a scheduler-oblivious synchronization algorithm achieves the highest throughput on a multiprogrammed ccNUMA multiprocessor.

A third motivation of this paper is the need to investigate whether alternative forms of synchronization for shared-memory multiprocessors can overcome the performance bottlenecks imposed by traditional synchronization algorithms, such as spin locks and barriers. Lock-free synchronization is the alternative synchronization methodology that has attracted the most considerable attention from researchers, both as an efficient alternative to spin locks and as a robust synchronization method for multiprogrammed execution environments.[22–27] The relative performance of lock-free synchronization compared to lock-based synchronization has not been quantified on ccNUMA multiprocessors yet.

1.2. Contributions

The paper presents a thorough evaluation of synchronization algorithms on a popular commercial ccNUMA multiprocessor, the SGI Origin2000.[28] Our evaluation is performed from the perspectives of the architecture and the operating system in a unified framework. Investigating the sources of synchronization overhead from both perspectives is necessary to develop synchronization algorithms with robust performance in both dedicated and multiprogrammed execution environments. The experiments presented in this paper and the associated methodology investigate the following issues:

- The performance implications of the ccNUMA cache coherence protocol on the latency and scalability of synchronization primitives.
- The impact of the architectural implications of ccNUMA and the implementation of synchronization primitives on the relative and the absolute performance of synchronization algorithms.

- The effectiveness of using specialized synchronization hardware to accelerate synchronization primitives and how can this hardware be leveraged in practical implementations of synchronization algorithms.

- The performance trade-off's of scheduler-conscious synchronization algorithms and synchronization-conscious scheduling strategies on a multiprogrammed ccNUMA multiprocessor.

- The performance of lock-free synchronization compared to lock-based synchronization, in terms of latency and waiting overhead.

The focal point of this paper is not the relative performance of different synchronization algorithms that implement the same synchronization semantics. We rather emphasize the implications of the architecture and the operating system on the absolute performance of synchronization algorithms.[7] Along with visiting each of the aforementioned issues, the paper contributes a new methodology for implementing fast locks, barriers and lock-free synchronization algorithms on ccNUMA multiprocessors.[29] Another contribution of the paper is the investigation of the mutual influence between synchronization and scheduling in a unified environment, which captures user-level, kernel-level and intermediate implementations of synchronization algorithms and scheduling strategies. This enables us to extract clean conclusions and propose guidelines for tuning the implementation of synchronization algorithms for multiprogrammed multiprocessors.

1.3. Methodology

The paper follows a bottom-up evaluation methodology. We first evaluate alternative implementations of elementary atomic synchronization instructions—also known as atomic Read-Modify-Write (RMW) instructions—and assess the performance implications of a ccNUMA directory-based cache-coherence protocol on the scalability of these instructions. In the same experiment we evaluate the effectiveness of using specialized synchronization hardware to speed-up the execution of RMW instructions under contention.

We then evaluate three sets of algorithms for spin locks, barriers and lock-free queues and explore the feasibility of leveraging specialized synchronization hardware to provide faster implementations of these algorithms. This leads us to a methodology for implementing *hybrid* synchronization algorithms, that is, algorithms that leverage both specialized synchronization hardware and the cache coherence protocol in software, to provide faster synchronization for moderate and large-scale ccNUMA systems. As

a special case study, we compare scalable algorithms for spin locks against lock-free algorithms that implement the same functionality. The evaluation of synchronization algorithms is conducted with microbenchmarks and complete parallel applications. The microbenchmarks measure the explicit and implicit latency of synchronization algorithms, while the application benchmarks investigate whether faster implementations of synchronization algorithms can provide sizeable performance improvement under realistic conditions.

In the final step of the evaluation procedure, we compare combinations of waiting algorithms, scheduler-conscious synchronization algorithms and operating system scheduling strategies, using multiprogrammed workloads that consist of multiple instances of complete parallel applications and sequential background load.

1.4. Outline

The remainder of the paper is organized as follows. Section 2 outlines the architecture of the Origin2000, emphasizing the architectural support for synchronization. Section 3 overviews the synchronization algorithms that we evaluate in this paper and alternative implementations of these algorithms. Section 4 presents experimental results that compare the performance of alternative implementations of spin locks, barriers, and lock-free algorithms. Section 5 presents results that evaluate combinations of waiting and scheduler-conscious synchronization algorithms with operating system scheduling strategies in a multiprogrammed execution environment. Section 6 concludes the paper.

2. THE SGI ORIGIN 2000

This section summarizes the fundamental architectural features of the SGI Origin2000, the ccNUMA multiprocessor platform on which we experimented for the purposes of this paper. The Origin2000 hardware support for synchronization is described in more detail in Section 2.2.

2.1. Architectural Overview

The SGI Origin2000[28] is a leading commercial ccNUMA multiprocessor introduced in October 1996. The system employs an aggressive memory and communication architecture to achieve high scalability. The building block of the Origin2000 is a dual-processor node. The node consists of 2 MIPS R10000 processors, with 32 Kbytes of split primary cache

and 4 Mbytes of unified second level cache per processor. Each node contains up to 4 Gbytes of DRAM memory, its corresponding directory memory and connections to the I/O subsystem. The components of a node are connected to a hub, which is in turn connected to a six-ported router. The hub serves as the communication assist of the node and implements the cache coherence protocol. The routers of the system are interconnected in a fat hypercube topology. The Origin2000 uses a flat memory-based directory cache coherence protocol which supports sequential consistency.[28] Along with several aggressive hardware optimizations for scaling the system's bandwidth, the Origin2000 provides software prefetching at the microprocessor level and hybrid hardware/software support for page migration and replication in the operating system, to achieve reasonable amounts of data locality in user programs without programmer intervention. The ratio of remote to local memory access time on the Origin2000 is no more than 3:1 for configurations of up to 128 processors. The Origin2000 runs Cellular IRIX, a distributed, SVR4-compliant 64-bit operating system with additional support for sophisticated NUMA-sensitive memory allocation and process scheduling.

2.2. Support for Synchronization

The Origin2000 provides two hardware mechanisms for implementing synchronization primitives in shared memory. The first mechanism uses microprocessor support for implementing atomic RMW instructions on cache-coherent volatile variables. The second mechanism uses specialized hardware which implements atomic RMW instructions directly at the memory modules of the system with a single round-trip message. These two mechanisms are outlined in the following paragraphs.

The MIPS R10000 ISA includes a load linked-store conditional (LL-SC) instruction, which is executed with hardware support at the microprocessor's cache controller. The load-linked reads a synchronization variable from its memory location into a register. The matching store conditional attempts to write the possibly modified value of the register back to its memory location. Store conditional succeeds if no other processor has written to the memory location since the load linked completed and fails if the cache line that stores the synchronization variable is by any means invalidated. A successful load linked-store conditional sequence guarantees that no conflicting writes to the synchronization variable intervene between the load linked and the store conditional. LL-SC is a versatile synchronization primitive, which can implement several atomic RMW instructions such as fetch&ϕ, test&set and compare&swap. [Note: Compare&swap takes three arguments, a pointer to a memory location, an expected value

and a new value. The instruction checks if the content of the memory location is equal to the expected value. If it is, the new value is stored in the memory location. The instruction returns an indication of success or failure.]

The implementation of LL-SC on the R10000 uses a reservation bit per cache line. This bit is set whenever the load linked is executed and invalidated before the execution of the store conditional, if the associated cache line is invalidated due to an intervening write from another processor, a context switch, or an exception. The load linked requests the cache line in shared state. If the matching store conditional succeeds, invalidations are sent to all the active sharers of the cache line. If the store conditional fails, no invalidations are sent out. A successful LL-SC sequence may incur two cache misses, which is likely to happen more frequently if several processors contend for the synchronization variable. On a ccNUMA architecture, most of these coherence misses are satisfied remotely through the interconnection network, rather than from the directory of the node with the processor that experiences the miss. The service of a remotely satisfied cache miss to dirty data requires up to 5 non-overlapped network messages and has the highest latency among all types of memory transactions on a ccNUMA system. If multiple processors issue LL-SC instructions to the same variable at the same time, the number of nonoverlapped network messages required to satisfy the coherence misses grows quadratically to the number of contending processors.

The second hardware synchronization mechanism of the Origin2000 implements atomic read, write and RMW instructions directly at the memory modules of the nodes. These instructions are issued as uncached loads and stores with arguments that encode the type of atomic operation to be applied in the specified memory location. At-memory uncached instructions are called *fetchops* in the Origin2000 terminology. Fetchops operate on 64-byte memory blocks which are not replicated in the processor caches. A fetchop is executed with a single round-trip network message and does not generate further network traffic as the operation bypasses completely the cache coherence protocol. Therefore, fetchops are appropriate to implement atomic RMW instructions on contended synchronization variables. A drawback of fetchops is the latency of reading an uncached variable, since fetchop read operations are always directed to a, possibly remote, memory module. The architects of the Origin2000 have circumvented this problem by adding a 1-entry fetchop cache at the memory interface of the hubs, to hold the most recently used fetchop variable.[30] The use of this cache reduces the best-case latency of a fetchop down to approximately the latency of a secondary cache hit, if the fetchop is satisfied from local memory. A second drawback of fetchops is that they

lack a powerful atomic primitive, such as compare&swap, or the atomic exchange of the contents of two memory locations. The absence of such a primitive is an important limitation, because it precludes the implementation of nonblocking synchronization algorithms. We elaborate this issue further in Section 3.

3. SYNCHRONIZATION ALGORITHMS

We consider the three most representative and widely used classes of synchronization algorithms for shared-memory multiprocessors, namely spin locks, barriers and lock-free data structures.[30] We draw the most popular algorithms from these classes to explore the performance trade-off's of synchronization on ccNUMA multiprocessors. In this section, we overview the algorithms that we evaluate and examine in more detail alternative implementations of the algorithms on the Origin2000.

3.1. Synchronization Algorithms for Parallel Programs

Spin locks are used to implement mutual exclusion on a shared memory multiprocessor. Each processor which attempts to enter a critical section acquires a lock and waits until the lock is released by the previous lock holder. In this paper, we evaluate the performance of the simple test&set lock with exponential backoff,[31] the ticket lock with proportional backoff[5] and the array-based queue lock.[3] The test&set lock scales poorly under contention because it does not limit the amount of RMW instructions required to acquire a lock. On a ccNUMA architecture, this results to exponential growth of network traffic incurred from coherence transactions on the synchronization flag. The ticket algorithm and the queue algorithm scale better, since they require a fixed number of atomic RMW instructions to acquire and release the lock.

A barrier is a global synchronization point, used to separate execution phases of a parallel program. The two types of barriers that have been most widely studied in the past are counter-based barriers and tree-based barriers. Counter-based barriers are simple to implement but may suffer from contention when multiple processors arrive at the barrier simultaneously. Tree barriers alleviate contention by propagating arrivals and departures from the barrier through a spanning tree that models the system topology. In this study we evaluate a simple counter barrier with local sensing and the MCS tree barrier.[5]

Lock-free synchronization replaces mutual exclusion in operations that require the atomic update of simple data structures, such as stacks and

linked lists. The notable advantage of lock-free synchronization compared to mutual exclusion is that it allows parallelism to be exploited by over-lapping the updates to shared data and nullifying the waiting latency when multiple processors attempt to access data with a lock. Lock-free syn-chronization has received considerable attention recently and is already applied in many practical applications.[22, 24] An important property that makes lock-free synchronization an attractive alternative is that under certain circumstances, lock-free synchronization can resolve contending processors in a wait-free, non-blocking fashion.[23, 32] The latter property makes lock-free synchronization a strong competitor to lock-based syn-chronization on multiprogrammed shared-memory multiprocessors, in which unpredictable preemptions of threads may affect dramatically the performance of synchronization algorithms.[4, 18, 25] In this paper we evaluate three lock-free, non-blocking algorithms that implement concurrent FIFO queues[26, 27, 33] and a simpler algorithm that implements a concurrent FIFO queue with fetch&add, without guaranteeing the nonblocking and wait-free properties.[7]

Table I summarizes the synchronization algorithms that we evaluate in this paper, along with the most relevant implementation details. The columns of the table from left to right report: The *cacheability* of the syn-chronization variables (acquire and waiting flags), that is, whether the syn-chronization variable is allocated in a memory line that can be replicated in the processor caches or not; the level of the memory hierarchy at which the RMW instructions used in the algorithms are implemented (cache or at-memory); the backoff strategy used in the algorithms (exponential or proportional), if any; and the level of the memory hierarchy where pro-cessors busy-wait mostly, when arriving at a synchronization point (cache or memory).

For each synchronization algorithm we attempted to have three imple-mentations. The first implementation uses volatile cached synchronization variables for acquiring and waiting and the LL-SC instruction of the MIPS R10000 to implement atomic RMW sequences. The second implementation uses uncached acquiring and waiting variables and the Origin2000 fetchops, in order to implement the atomic RMW instructions directly at-memory and reduce the coherence-induced overhead of synchronization. In this imple-mentation, processors busy-wait by polling a single uncached waiting flag that resides in the same memory module with the acquire flag. The third implementation is a *hybrid* implementation, in which the acquire flags are allocated in uncached memory locations and the waiting flags are allocated in cached memory locations.[29] This implementation tries to obtain the best of the two approaches, that is, busy-wait in the local cache and syn-chronize at-memory.

Table I.   Synchronization Algorithms Evaluated on the SGI Origin2000

			Locks		
Algorithm	Acquire flag	Waiting flag	RMW	Backoff	Spin
text&set, LL-SC	cached	cached	cached	exp	cache
text&set, at-mem	uncached	uncached	at-mem	exp	mem
text&set, hybrid	uncached	cached	at-mem	exp	cache
queue lock, LL-SC	cached	cached	cached	n/a	cache
queue lock, at-mem	uncached	uncached	at-mem	n/a	mem
queue lock, hybrid	uncached	cached	at-mem	n/a	cache
ticket lock, LL-SC	cached	cached	cached	prop	cache
ticket lock, at-mem	uncached	uncached	at-mem	prop	mem
ticket lock, hybrid	uncached	cached	at-mem	prop	cache
			Barriers		
counter, LL-SC	cached	cached	cached	n/a	cache
counter, at-mem	uncached	uncached	at-mem	n/a	mem
counter, hybrid	unchached	cached	at-mem	n/a	cache
MCS tree barrier	cached	cached	n/a	n/a	cache
			Lock-Free Queues		
Algorithm	Access points	Data	RMW	Backoff	
Michael, Scott	cached	cached	cached	exp	
Valois	cached	cached	cached	exp	
Prakash, Lee, Johnson	cached	cached	cached	exp	
fetch&add, LL-SC	cached	cached	cached	exp	
fetch&add, at-mem	uncached	uncached	at-mem	exp	
fetch&add, hybrid	uncached	cached	at-mem	exp	

Lock-free synchronization presents a special case for the hybrid implementation, in the sense that lock-free algorithms have no distinct busy-wait phase. Instead, lock-free algorithms use a finite number of atomic RMW instructions to update the access points to a lock-free data structure (e.g., the head and the tail of a queue) and a number of reads and writes to maintain linkage pointers in the data structure. In the case of lock-free algorithms, the hybrid implementation executes the atomic RMW instructions directly at the memory nodes, while the memory accesses that maintain linkage pointers and data are executed with regular cached loads and stores. The complete methodology for implementing hybrid synchronization primitives and transforming any synchronization algorithm into a hybrid one is presented elsewhere.[29]

In all cases, the backoff interval used in the algorithms was tuned after extensive experimentation. We tried to exploit all system features that could accelerate synchronization operations. The two most important optimizations in the implementations were the use of LL-SC as an implicit test&set, instead of a test&set function call and the exploitation of the fetchop cache of the Origin2000. The first optimization sped-up significantly the LL-SC implementation of the test&set lock. The second optimization improved the synchronization algorithms that use a single flag for both the acquire and the release, namely the test&set lock and the counter barrier.

We could not provide at-memory or hybrid implementations of certain synchronization algorithms. Locks and nonblocking lock-free algorithms that use linked lists and compare&swap or fetch&store could not be implemented due to the lack of hardware support for executing these RMW instructions directly at the memory modules of the Origin2000. This has proven to be the most restrictive limitation of the hardware platform on which we experimented. To circumvent this problem, we compared the array-based implementations of the queue lock and the fetch&add lock-free queue, against linked list-based implementations of the same algorithms with LL-SC. For the list-based implementations we used the algorithms that appear to be the fastest of their kind in the literature, namely the MCS queue lock,[5] Fu and Tzeng's queue lock[13] and Michael and Scott's non-blocking lock-free queue.[33] In all cases the array-based implementations performed marginally to moderately better, primarily due to the high cost of compare&swap on the Origin2000. We therefore believe that the aforementioned limitation does not affect our quantitative results.

An at-memory implementation of the array-based queue lock was not possible, because the system on which we experimented limited the number of uncached memory blocks that can be allocated in a program and prevented their distribution, so that each processor could have a private uncached waiting flag residing in a local memory module. Since localization of the waiting flags was not possible, the implementation of the queue lock with uncached instructions could not be customized to ensure a fair comparison.

3.2. Synchronization Algorithms and Scheduling Strategies for Multiprogrammed Execution Environments

The algorithms used to cope with asynchronous preemptions of synchronizing threads in programs executing on a multiprogrammed shared-memory multiprocessor can be categorized in two classes. The algorithms in the first class assume a standard time-sharing scheduler with priority control and embed in the synchronization operations either a waiting

algorithm that attempts to improve processor utilization,[17, 19] or an inter-face with the operating system used to prevent inopportune preemptions of threads or improve the throughput of synchronization operations in the presence of these preemptions.[18, 34] Waiting synchronization algorithms are frequently characterized as spin-block or two-level synchronization, in the sense that they usually apply a competitive algorithm that instructs each synchronizing thread to spin for a predefined amount of time before blocking to release the processor. The implementation of these algorithms does not require modifications to the operating system. Algorithms that use an interface with the operating system are characterized as either preemp-tion-safe or scheduler-conscious, depending on whether they avoid inopportune preemptions of threads or try to reschedule synchronization operations after detecting such preemptions. The shortcoming of these algorithms is that they require modifications to the operating system. In this paper we evaluate busy-wait, block, spin-block and scheduler-conscious synchroniza-tion algorithms in the same framework.

The second class of multiprogramming-conscious synchronization mechanisms includes scheduling algorithms that take into account the fact that the threads of parallel programs may synchronize frequently. The fundamental idea of these algorithms is to coschedule the threads of the same parallel program as a gang,[21] in order to ensure that synchronizing threads are always running together. Gang scheduling is a well-known technique for tuning the performance of programs with fine-grain syn-chronization on nondedicated multiprogrammed environments. In the con-text of this paper, gang scheduling is evaluated against different waiting algorithms that operate with a time-sharing scheduler.

Along with multiprogramming-conscious synchronization algorithms and synchronization conscious scheduling algorithms, we also evaluate lock-free synchronization algorithms as an alternative for enhancing the performance of parallel programs on multiprogrammed shared-memory multiprocessors. Recall that several lock-free algorithms satisfy the non-blocking and wait-free properties, thus eliminating the waiting overhead exposed at synchronization points due to preemptions of synchronizing threads.

Table II summarizes the synchronization algorithms for multipro-grammed environments with which we experimented. We have made an effort to implement and compare the algorithms under the same conditions. The three waiting algorithms (busy-wait, spin-block and block) are straight-forward to implement at user-level, by embedding busy-wait loops and the self-blocking system call of IRIX in the synchronization primitives. The waiting interval in the spin-block algorithm is selected to be roughly equal to twice the cost of a context switch for both locks and barriers.[17] In the

Table II.   Synchronization Algorithms for Multiprogrammed Execution Environments

Multiprogramming-conscious synchronization algorithm	Algorithms applied to
busy-wait	locks, barriers
competititve spinning (spin-block)	locks, barriers
block	locks, barriers
scheduler-conscious synchronization	locks, barriers
lock-free synchronization	access to FIFO queues
gang scheduling	locks, barriers

case of scheduler-conscious barriers, the waiting interval is set to be proportional to the number of processors that have not arrived at the barrier yet.[18] Lock-free synchronization is also implemented at user-level, however it can only be compared against spin lock algorithms that implement the same semantics, i.e., protect the same concurrent data structure.

Gang scheduling is provided as a special option for scheduling the threads of a parallel program in IRIX, however this feature is obsolete in the latest versions of the operating system and our earlier results with gang scheduling in IRIX were not interpretable.[7] For this reason, we implemented an emulator of gang scheduling at user-level using the following methodology. We devote one processor of the system to run a user-level CPU allocator,[35] the purpose of which is to block and unblock the threads of a parallel program as a unit. The CPU allocator communicates with parallel programs via memory-mapped virtual memory segments and uses the IRIX blockproc/unblockproc interface[36] to explicitly schedule threads, under the restriction that these threads have the same user and group ID with the CPU allocator. In our experiments with multiprogrammed workloads, each program requests one processor less than the total number of processors in the system. We used the same CPU allocator to implement a scheduler-conscious version of the queue lock. In this case, the CPU allocator emulates a standard UNIX-like time-sharing scheduler, which in addition, informs parallel programs of thread preemptions. This information is used in order to avoid passing locks to preempted threads, as described by Kontothanassis *et al.*[18]

4. SCALABILITY OF SYNCHRONIZATION PRIMITIVES AND ALGORITHMS

In this section we examine in detail the implications of the implementation of atomic RMW instructions on the performance of synchronization

algorithms. Although the results provide useful insights on the relative performance of different synchronization algorithms that implement the same semantics, the focal point is the investigation of the extent up to which additional hardware support can deliver faster synchronization on ccNUMA multiprocessors and the conditions under which this support can be leveraged to enhance the performance of parallel programs. The relative performance of algorithms for locks and barriers is commented briefly, as the overall results do not differ significantly from those extracted in previous studies.[5] More emphasis is given on the relative performance of lock-free synchronization, as an alternative to lock-based synchronization. Section 4.1 outlines our evaluation methodology. Sections 4.2–4.6 present results from experiments with microbenchmarks and parallel applications from the SPLASH-2 and SPEC benchmark suites, the performance of which is sensitive to the overhead of synchronization.

4.1. Methodology

Microbenchmarks were the common practice for evaluating synchronization algorithms in earlier works.[3, 5] However, several recent studies have doubted the methodological validity of microbenchmarks and based their experiments on complete parallel applications.[6, 8, 9] We believe that microbenchmarks are useful as an evaluation tool, due to their inherent ability to control the execution environment and isolate performance characteristics. The shortcoming of microbenchmarks is that their results do not always agree with the performance trends observed in real applications. This is particularly true for synchronization, as the impact of a synchronization algorithm on the performance of a parallel program depends on parameters orthogonal to the algorithm, such as the frequency of synchronization operations, the degree of contention at synchronization points and the load imbalance of the parallel computation.[8]

We evaluate synchronization algorithms using both microbenchmarks and real applications. We used a fully parameterized microbenchmark, in which the degree of contention at synchronization points, the average length of the critical section, the average length of the delay interval between successive synchronization periods and the probabilistic load imbalance in this interval are input parameters. The microbenchmark used in our experiments is shown in Fig. 1. The synchronize() function may be any of a lock/unlock pair enclosing a critical section with an average sequential run length equal to cslength, a barrier, or a sequence of accesses to a lock-free queue with an average sequential run length equal to cslength. Inside critical sections and during the delay interval, each processor modifies a shared array of uncached integers with remote memory

```
barrier();
for (i = 0; i < num_periods/num_processors); i++) {
    /* Each processor performs dummy computation for a delay period of
       a fixed average length with probabilistic load imbalance. The
       processor issues uncached instructions during the delay period. */
    delay(length, imbalance);
    /* Synchronize with other processors. In case the synchronization
       operation is a lock, execute a critical section of an average
       length equal to cslength. */
    synchronize(cslength);
}
barrier();
```

Fig. 1.   Code of the synchronization microbenchmark.

accesses. The choice of accessing remote memory during delay intervals and critical sections is made to evaluate the impact of coherence traffic incurred from synchronization, on the latency of memory accesses performed during the execution of useful computation that proceeds concurrently with synchronization. By varying the parameters in the microbenchmark it is possible to run experiments that achieve a good coverage of the synchronization patterns of real applications with fine-grain parallelism.

We evaluated the implementations of synchronization algorithms using also four real programs with high synchronization overhead, namely Radiosity, Barnes and LU from the SPLASH-2 benchmark suite,[37] and HYDRO2D from the SPECcpu95 benchmark suite.[38] All four applications have proven to be sensitive to the implementation of their synchronization constructs on ccNUMA multiprocessors.[8, 39] Barnes and Radiosity are lock-intensive programs, while LU and HYDRO2D have frequent barriers.

Our metric for evaluating synchronization algorithms is the average execution time of a synchronization period, including the delay interval between successive synchronization periods. This metric provides a more concise view of the performance of a synchronization algorithm compared to metrics such as the average time per lock, or the average delay in the critical section.[6] It accounts for the overhead of the actual synchronization algorithm and the interference between the synchronization algorithm and the computation performed by processors during delay intervals and critical sections. In the case of parallel applications, we use the absolute speedup as the performance metric. All the experiments were executed on a 64-processor SGI Origin2000, with MIPS R10000 processors clocked at 250 MHz. The system ran IRIX 6.5.4 during the experiments.

4.2. Scalability of Atomic RMW Instructions

The synchronization algorithms considered in this paper, with the exception of tree barriers, use atomic RMW instructions to resolve

contending processors. Section 2 elaborated that an implementation of these instructions with LL-SC may not be scalable because under contention, an LL-SC sequence is likely to incur two cache misses, which are in most cases satisfied remotely. A failed LL-SC sequence may cost as much as the cost of sending 10 nonoverlapped messages between nodes. On the contrary, if the atomic instruction is executed with special hardware at the memory modules, it requires only 2 nonoverlapped messages to and from the memory module that stores the synchronization flag. Intuitively, when all processors of the system contend at a synchronization point, the throughput of the at-memory RMW instructions can be as much as 5 times the throughput of RMW instructions implemented with LL-SC. Figure 2 verifies precisely this intuition. The left chart shows the average execution time of two implementations of fetch&add, with LL-SC and at-memory, on a progressively increasing number of processors that execute the atomic instruction concurrently. It is evident that atomic RMW instructions implemented at-memory reduce drastically the overhead of synchronization primitives.

The right chart in Fig. 2 illustrates the shortcoming of implementing a synchronization algorithm using only uncached variables. The chart shows the results of an experiment in which the processors use the ticket lock to repeatedly access a nonempty critical section with a sequential run length equal to $7\,\mu s$ and then wait for a nonempty delay interval with a sequential run length equal to $25\,\mu s$. The probabilistic load imbalance in the delay interval is $\pm 10\%$. In this experiment, the lock acquire is implemented with an implementation of fetch&add at-memory and the release is implemented alternatively with a cached or an uncached waiting
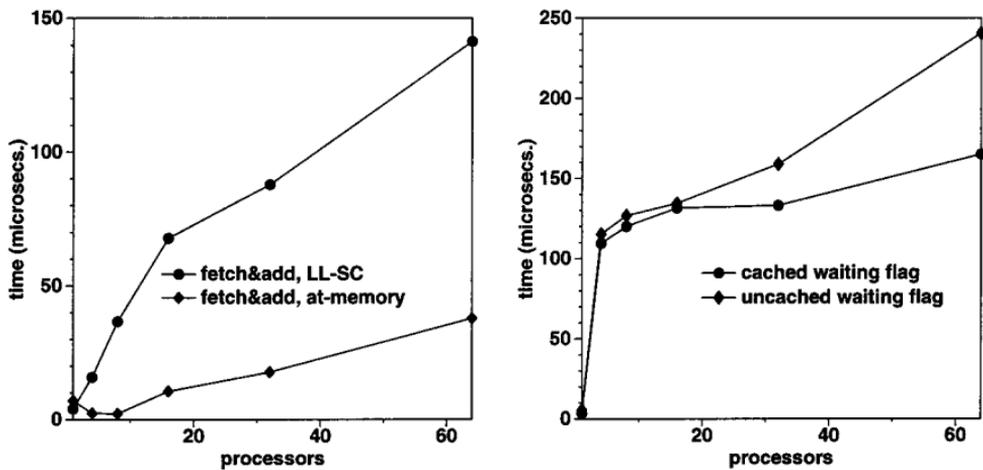

Fig. 2. Scalability of atomic RMW instructions (left) and the impact of busy-waiting on uncached variables (right).

flag. The result shows that busy-waiting on an uncached flag incurs excessive network traffic and may dilate significantly the memory accesses performed during critical sections and delay intervals. Note that the delay interval may proceed in parallel with the critical section due to load imbalance. In this case, using an uncached flag in the busy-wait loop increases the execution time of the synchronization period by as much as 45%, when all 64 processors contend to access the critical section.

4.3. Spin Locks

Figure 3 illustrates the average execution time of a synchronization period, computed from an experiment in which the microbenchmark of Fig. 1 is executed with all processors contending to access a critical section with spin locks. The microbenchmark was executed with two different sets of parameters for the length of the critical section and the delay interval. The execution of the benchmark was repeated several times to ensure numerical stability in the results. The computed time is the average of 10 executions with 1000 synchronization periods each. In the left chart, the critical section and the delay interval are empty, therefore the chart compares the net overhead of the algorithms. In the right chart, both the critical section and the delay interval are not empty. Furthermore, artificial load imbalance of $\pm 10\%$ is introduced in the delay interval. The microbenchmark in this case simulates a program with fine-grain synchronization and real computation with poor cache locality, to assess the indirect effects of synchronization algorithms on the computation performed inside
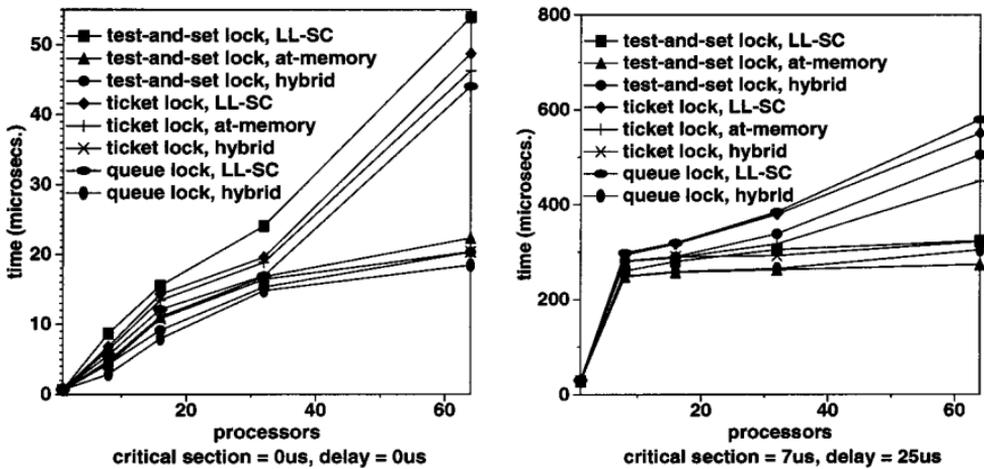

Fig. 3.   Comparison of spin locks with microbenchmarks.

and outside critical sections during synchronization periods. The synchronization primitives dilate significantly the execution of useful computation, since they introduce network traffic and contention at the memory modules at which the synchronization flags are stored.

The left chart in Fig. 3 shows that when all three locks are implemented with LL-SC, the queue lock is faster than the ticket lock, which is in turn faster than the test&set lock. This result simply generalizes previous results on the relative scalability of spin lock algorithms on the ccNUMA architecture. Notice however that the three implementations of the locks with at-memory RMW instructions are moderately or significantly faster than the respective LL-SC implementations. Interestingly, the at-memory ticket lock and the at-memory queue lock are significantly slower than the at-memory test&set lock. The reason is the fetchop cache of the Origin2000 nodes. This cache holds at most one uncached synchronization variable and is sufficient for the test&set lock, which uses the same variable for both the acquire and the release. The ticket lock and the queue lock use separate variables for acquiring and busy-waiting and these variables cannot be held simultaneously in the fetchop cache. The hybrid implementations of the three locks have the lowest overhead due to the combined effect of eliminating the coherence overhead of RMW instructions and reducing the network overhead of the busy-wait phase, by having the waiting flags replicated in the caches.

The right chart in Fig. 3 illustrates results from an experiment, in which the critical sections and the delay intervals between them include some amount of useful computation. The critical observation is that the at-memory implementations suffer from the effects of network traffic and perform worse than the LL-SC implementations. This happens because the computation in the delay interval and the critical section is extremely sensitive to the network traffic generated from uncached instructions when processors busy-wait. Note that this is not the case if the delay interval consists of accesses to cacheable data. In that case, the at-memory implementations of the locks outperform the LL-SC implementations. The uncached memory accesses executed in the microbenchmark represent a worst-case scenario for at-memory synchronization instructions. In this experiment, the hybrid ticket lock and the hybrid queue lock appear to be the fastest. A second-order conclusion drawn from the chart is that the scalability of the queue lock shows up under more realistic execution conditions.

Figure 4 illustrates the speedups of the SPLASH-2 Radiosity and Barnes benchmarks with alternative implementations of the spin locks used in the benchmarks. Radiosity was executed with the default input set and Barnes was executed with an input set of 16K bodies. The results for
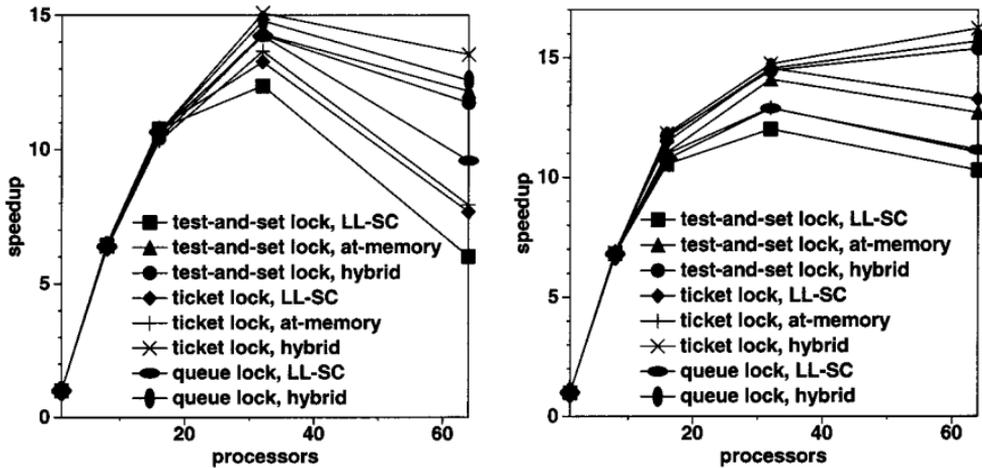
Fig. 4.   Speedup of the SPLASH-2 Radiosity (left) and Barnes (right) benchmarks with different implementations of spin locks.

Radiosity agree qualitatively with the results in the left chart of Fig. 3. The LL-SC locks have identical relative performance in both experiments and the at-memory locks outperform the LL-SC locks. This implies that the traffic generated by the at-memory locks does not interfere significantly with memory accesses during the execution of the benchmark. The hybrid locks are comparable to the at-memory locks and the hybrid queue lock delivers the best speedup on 32 and 64 processors. The results for Barnes agree mostly with the results in the right chart of Fig. 3. Barnes is more sensitive to the network traffic generated by the at-memory read instructions issued during the busy-wait phase of the locks. The hybrid implementations of the locks achieve by far the best speedup for the given problem sizes. The at-memory ticket and the at-memory queue lock perform worse than the respective LL-SC implementations, while the at-memory test&set lock performs better than the LL-SC implementation of the same lock.

4.4. Lock-Free Algorithms

Figure 5 depicts the performance of alternative implementations of lock-free FIFO queues. The results are obtained from 10 executions of the microbenchmark, in which every processor executes 1000 back-to-back enqueue and dequeue operations (left chart) or enqueue and dequeue operations with an average sequential run length of $40\,\mu$s, separated by unbalanced delay intervals with an average sequential run length of $5\,\mu$s (right chart). Note that the lock-free queue exploits parallelism from concurrent updates in the second case.
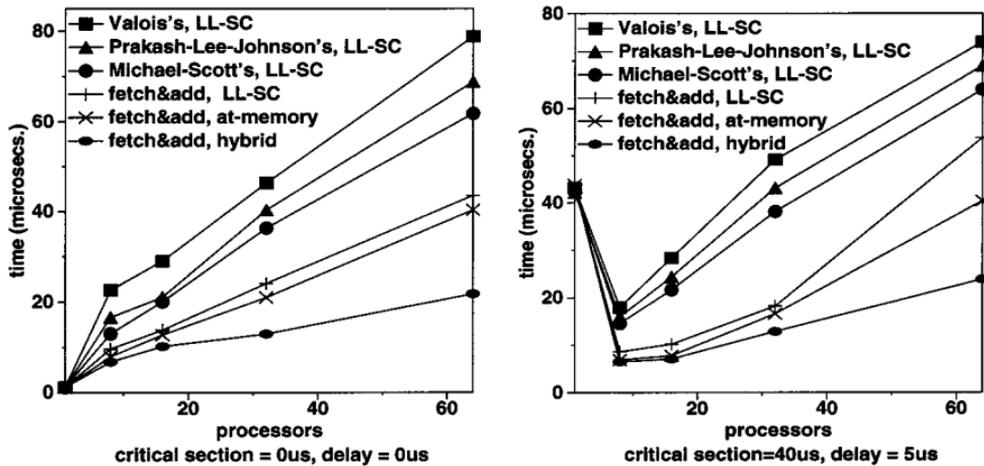
Fig. 5. Comparison of lock-free FIFO queues with microbenchmarks.

The three nonblocking FIFO queue algorithms have higher latency compared to the fetch&add algorithm. There are two reasons that explain this behavior. The first is that nonblocking algorithms use several expensive compare&swap instructions on the critical path of the enqueue and dequeue operations. Compare&swap can only be implemented with LL-SC on the Origin2000 and scales poorly under contention. Since compare&swap can not be implemented with fetchops, the nonblocking algorithms can not take advantage of the fast uncached synchronization instructions. The second reason that limits the scalability of nonblocking algorithms is their complex memory management schemes. Each of the three nonblocking algorithms has customized memory management code in order to avoid the ABA problem, i.e., the situation in which a pointer to a node in the queue is read by one processor, then recycled by another processor and then modified again by the first processor. Although the compare&swap of the first processor succeeds and completes the update, the queue becomes corrupted after this operation. Avoiding the ABA problem requires additional book-keeping code that tracks the reference history of each pointer. The differences in the performance of the three nonblocking algorithms are actually attributed to the memory management code. Valois's algorithm uses a fairly complex memory management scheme to handle many corner cases. Prakash *et al.*'s scheme[26] is simpler and Michael and Scott's scheme[33] is even simpler, although the latter does not ensure correctness when the same processor issues more than one consecutive enqueue or dequeue operations. We fixed this bug for the purposes of the evaluation.[7]

The lock-free queues with fetch&add are clearly faster than the non-blocking algorithms, although they compromise the nonblocking property.

Among the alternative implementations of the fetch&add algorithm, the at-memory implementation is moderately to significantly faster than the LL-SC implementation and the hybrid implementation is significantly faster than both the at-memory and the LL-SC implementation. The at-memory implementation is better than the LL-SC implementation, because the critical path of the lock-free algorithm consists of a finite number of fetch&add instructions and reads and writes to the queue node pointers. The at-memory implementation is not prone to the problem of network traffic because the lock-free algorithm does not busy-wait. However, notice that it is still advantageous to execute the read and write operations of the lock-free algorithm with regular cached instructions.

We did not test the lock-free algorithms with an application benchmark, because the FIFO queuing algorithm is not suitable for any of the application benchmarks with which we experimented. Some SPLASH-2 benchmarks that use shared work queues (e.g., Radiosity), require that enqueuing and dequeuing can be performed to and from both ends of the queue. We were not able to find in the literature or devise a lock-free algorithm that meets these criteria. The best approximation we came up with is a lock-free, nonblocking LIFO queue (i.e., a stack) implemented with compare&swap. We compared the performance of the Raytrace benchmark with a nonblocking LIFO work queue to the performance of the same benchmark with a LIFO work queue protected by the hybrid queue lock. The nonblocking queue achieved a moderate improvement of speedup, equal to 9.2% on 16 processors and 3.1% on 32 processors. On 64 processors, the speedup obtained with the nonblocking queue was 1.7% lower compared to the speedup obtained with the hybrid queue lock.

4.5. Lock-Based vs. Lock-Free Synchronization

In order to compare lock-free against lock-based synchronization primitives we modified the microbenchmark so that in the synchronize( ) block, each processor is either executing consecutive enqueue and dequeue operations to a lock-free FIFO queue, or the same number and type of operations to a sequential pointer-based FIFO queue protected with spin locks. For the purposes of the comparison, we used the most efficient lock algorithm based on our experimental evidence, i.e., the hybrid queue lock, the most efficient nonblocking lock-free queue, i.e., Michael and Scott's queue and the fastest implementation of the FIFO lock-free queue with fetch&add. Figure 6 illustrates the results of this experiment.

The results demonstrate an interesting trade-off. The lock-free queues outperform the queue lock when the synchronization instructions used in the algorithms are implemented with LL-SC. Witness however that when
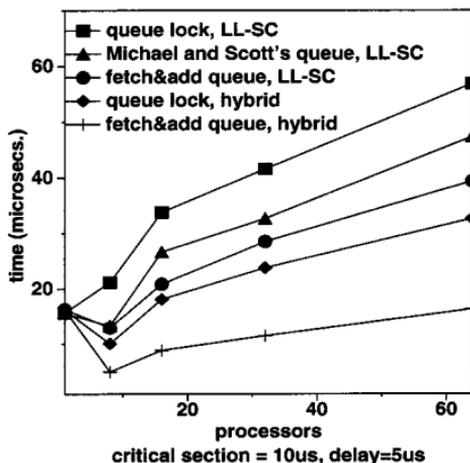
Fig. 6. Comparison of lock-free and lock-based synchronization algorithms.

the queue lock is implemented with at-memory RMW instructions and cached waiting flags, it outperforms the implementations of lock-free queues with LL-SC. This means that the architectural implications on the scalability of RMW instructions can affect the relative performance of lock-free and lock-based synchronization algorithms. In the case of the Origin2000, the nonblocking implementations of lock-free queues do not outperform the hybrid queue lock, because the nonblocking queues require several expensive compare&swap instructions, which can only be implemented with LL-SC. Similar conclusions are drawn by comparing the LL-SC implementation of the fetch&add lock-free queue to the hybrid queue lock. The hybrid fetch&add queue on the other hand performs considerably better compared to the other implementations. The results from this experiment are in many aspects instrumental. Lock-free synchronization is preferable to lock-based synchronization, under two conditions: first, the lock-free algorithm can replace the lock-based algorithm without affecting correctness; and second, the lock-free algorithm can be implemented with atomic RMW instructions which are at least as scalable as the RMW instructions used in the lock-based algorithm.

4.6. Barriers

Figure 7 compares alternative implementations of a barrier. The results are obtained from executing two versions of the microbenchmark. The first version executes back-to-back barriers in the synchronize() block, while the second executes consecutive barriers separated by unbalanced delay intervals with a sequential run length of $50\,\mu$s. The
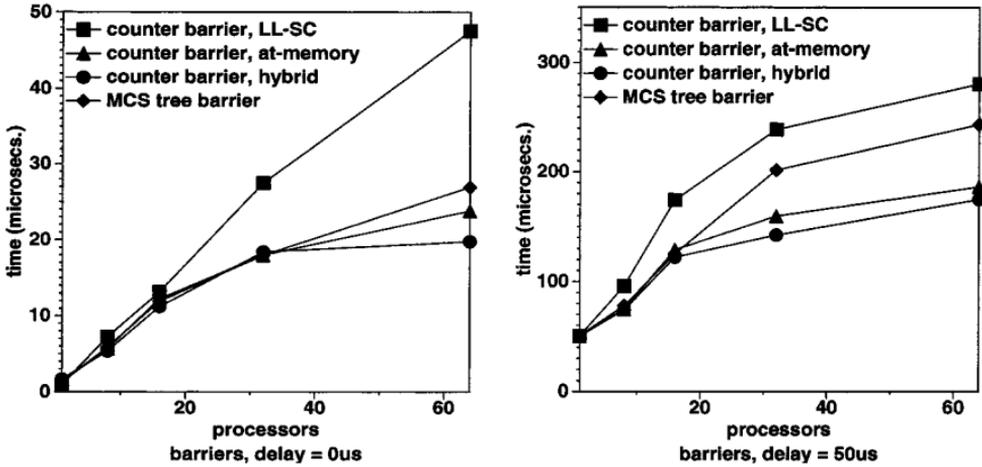
Fig. 7.   Comparison of barriers with microbenchmarks.

probabilistic load imbalance is set equal to $\pm 10\%$. As in the case of locks, the computation in the delay intervals contains uncached accesses to shared data. The results show that the simple counter barrier is not scalable when the atomic increment of the counter is implemented with LL-SC, but is the fastest when the atomic increment is implemented at-memory. In the at-memory implementation of the counter barrier, the waiting flag can be cached in the fetchop cache of the Origin2000 nodes. This effect helps the at-memory counter barrier to perform almost as good as the hybrid counter barrier.

The at-memory and the hybrid counter barriers outperform the tree barrier, a result which is in contrast with previous results obtained from experiments on noncache-coherent shared-memory multiprocessors.[5] On ccNUMA architectures, the hot spot in the busy-wait phase of the counter barrier is not as severe as on noncache-coherent systems, because the waiting flag can be cached without generating network traffic, until the last processor arrives at the barrier. On the other hand, the tree barrier has additional overhead in the arrival and the wakeup phases, which is not outweighed by the reduction of coherence traffic from local spinning and reduced sharing. Other barriers that we tested on the Origin2000, such as the dissemination and the tournament barrier,[5] did not outperform the MCS barrier.

Figure 8 illustrates the performance of the the SPECfp95 HYDRO2D benchmark and the SPLASH-2 LU benchmark, with different implementations of the barriers used in the benchmarks. LU was executed with a problem size of 128×128. The results agree with the results in Fig. 7, however the differences between alternative implementations of the barrier
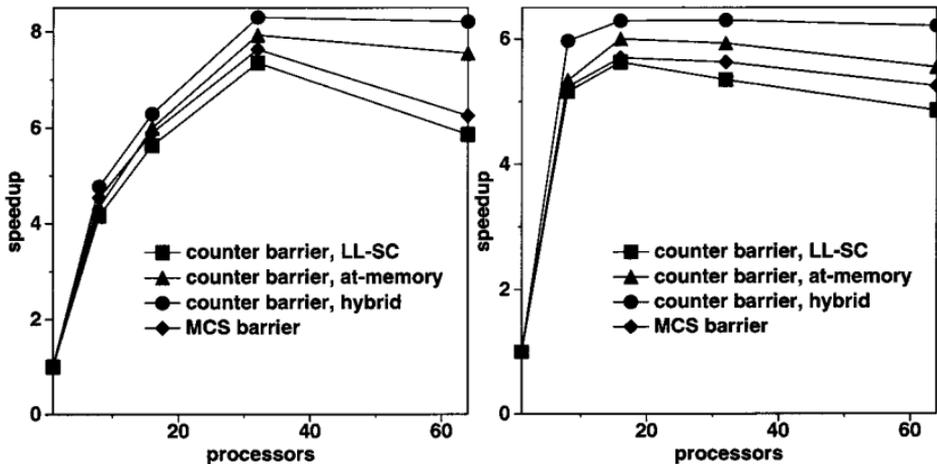
Fig. 8.   Speedups of SPEC HYDRO2D (left) and SPLASH-2 LU (right) with different implementations of the barrier.

tend to vanish when the problem size is increased. In that case, load imbalance is responsible for the most significant fraction of the synchronization overhead. Contention and network traffic due to synchronization instructions are alleviated when the frequency of barriers is reduced. HYDRO2D was parallelized with OpenMP directives[40] and executed with the reference problem size. The performance trends of different barrier implementations match those of LU and the microbenchmarks. HYDRO2D appears to be a little more sensitive to the implementation of the barrier on 64 processors. However this is a second-order effect since the speedup of the program flattens beyond 32 processors.

5. SYNCHRONIZATION ALGORITHMS FOR MULTIPROGRAMMED ENVIRONMENTS

5.1. Methodology

Evaluating multiprogrammed workloads with microbenchmarks is a daunting task, because the parameter space than needs to be explored in order to draw quantitative conclusions is huge. The reader can refer to an earlier paper[7] for a sample of the results obtained from executions of multiprogrammed workloads with our synchronization microbenchmarks. For the sake of brevity and clarity, in this paper we present only the results from experiments with multiprogrammed workloads consisting of complete parallel applications running together with sequential jobs that serve as background noise.

In order to experiment with the behavior of different synchronization algorithms for multiprogrammed environments, we executed multiprogrammed workloads consisting of two copies of the same parallel program and background noise. The background noise is produced from a repetitive invocation of the make utility, used to compile the files in the gcc compiler source tree. The invocations of make produce several hundreds of short-lived sequential processes that compete for CPU time with threads belonging to parallel programs, in a fairly unpredictable and random manner. These workloads model a moderately loaded multiprogrammed system operating in production mode, in which CPU-intensive parallel programs compete with jobs submitted from interactive users. The workloads are representative of the daytime load of contemporary ccNUMA servers.

In these experiments, parallel programs request 63 out of the 64 processors of the system. One processor is always set aside for running the CPU allocator, the module that implements gang scheduling and scheduler-conscious synchronization. The base time quantum in our emulation of gang scheduling is set to 100 ms, which is equal to the time quantum used by the IRIX earnings-based scheduler.[41] All the experiments with multiprogrammed workloads were repeated at least 10 times in order to ensure numerical stability in the results. The standard deviation ranges between 2–10%, which is acceptable given the dynamic characteristics of the workloads.

5.2. Synchronization Algorithms with Time-Sharing and Gang Scheduling

Figure 9 illustrates the normalized average execution time of Radiosity and HYDRO2D in the multiprogrammed workloads with three waiting algorithms executed in a time-sharing environment (busy-wait, block, and spin-block), scheduler-conscious synchronization in a time-sharing environment, and gang scheduling. Normalization is performed against the execution time of the benchmarks with time-sharing and busy-wait at all synchronization points. The waiting algorithms were embedded in the spin locks of Radiosity and the barriers of HYDRO2D. We used the hybrid implementation of the queue lock and the hybrid implementation of the counter barrier, since these implementations have proven to be the fastest in the experiments with standalone parallel benchmarks. Note that the waiting algorithms are used only in conjunction with time-sharing.

The scheduler-conscious lock is implemented as a hybrid queue lock, in which the thread that holds the lock scans the queue of waiting threads and passes the lock to the first nonpreempted waiting thread, thus sacrificing fairness for the sake of utilization. Preemptions are communicated to
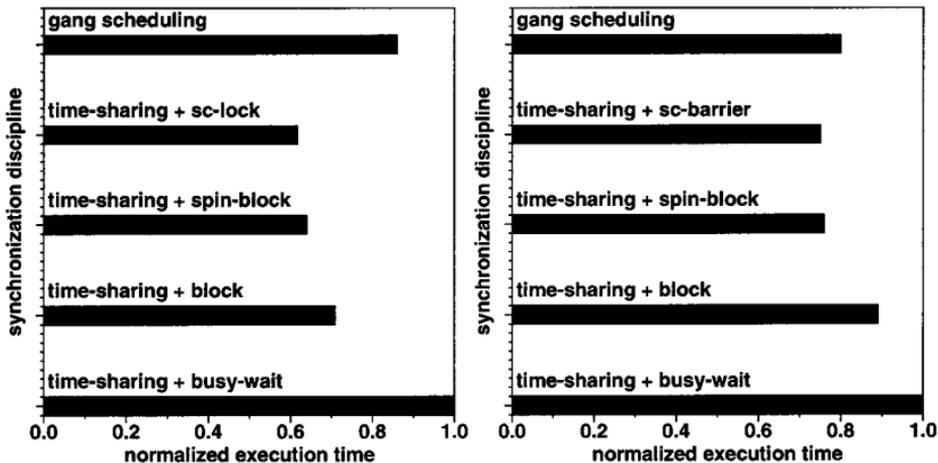
Fig. 9. Performance of waiting algorithms for time sharing schedulers and gang scheduling in Radiosity (left) and HYDRO2D (right).

the programs from the user-level CPU allocator through shared-memory. The scheduler-conscious barrier is equivalent to the barrier with competitive spinning in all aspects but the waiting interval before blocking. Each processor waits for a different amount of time, which is proportional to the number of processors that have not arrived at the barrier yet.

Gang scheduling improves the average turnaround time by 14% in Radiosity and 11% in HYDRO2D. Although gang scheduling is clearly advantageous compared to time-sharing when the programs in the workload busy-wait at synchronization points, its performance is inferior to that of time-sharing when the latter is combined with competitive spinning. Compared to busy-wait, competitive spinning improves performance by 24% in HYDRO2D and by as much as 36% in Radiosity. Time-sharing with blocking outperforms gang scheduling in Radiosity but underperforms gang scheduling in HYDRO2D.

In general, it is not clear whether gang scheduling is the best choice for tightly synchronized programs that execute on a multiprogrammed ccNUMA multiprocessor. We bear in mind that this result may be biased by the overhead of our CPU allocator, which actually emulates gang scheduling. We have made every effort to minimize this overhead, by having the CPU allocator communicate with parallel programs and the operating system through shared-memory. A second factor that may bias the results is the implication of the selected scheduling strategy on the memory system performance and in particular, the cache hit ratio. If the time-sharing scheduler uses affinity links,[42] the memory performance of parallel programs tends to improve because affinity scheduling takes into

account the cache footprint of each thread while assigning threads to processors. IRIX does make use of affinity links in its time-sharing scheduler. In order to ensure a fair comparison, we let IRIX use the same affinity links in our emulated gang scheduler. This means that during the execution of the workloads with both time-sharing and gang scheduling, the affinity links of IRIX determine the mapping of threads to processors. Therefore, the performance implications of affinity are expected to be uniform with both schedulers.

The scheduler-conscious lock improves marginally the performance of Radiosity compared to the best waiting algorithm, while the scheduler-conscious barrier performs exactly as well as the best waiting algorithm in HYDRO2D. The scheduler-conscious lock is susceptible to the overhead of the CPU allocator, while the scheduler-conscious barrier is not. The marginal improvements attained by scheduler-conscious synchronization do not justify its implementation cost. A fully customized implementation of scheduler-conscious synchronization at kernel level, including the management of thread priorities, may improve the picture.

5.3. Lock-Free Synchronization

In order to evaluate lock-free synchronization as an alternative to waiting synchronization algorithms and synchronization-conscious scheduling strategies, we compared an implementation of Radiosity with lock-free LIFO work queues to an implementation of Radiosity with LIFO work queues protected with spin locks. To implement the latter option, we modified the code of the benchmark, so that the original work queues that enabled enqueuing and dequeuing at both ends were replaced by stacks. As mentioned earlier, the original work queues could not be implemented in a lock-free fashion. Note that the use of LIFO work queues reduces the speedup of the benchmark, but ensures a fair comparison between lock-based and lock-free synchronization.

Figure 10 illustrates the results from this experiment. The performance of the lock-free LIFO queue is 34% better compared to time-sharing with busy-wait and 14% better compared to the scheduler-conscious queue lock. The result verifies the intuition that nonblocking synchronization is an effective and cheap way to increase throughput in a multiprogrammed environment, without reverting to implementations that interact directly with the operating system.

6. CONCLUSIONS

In this section, we summarize the conclusions drawn from the results presented in Sections 4 and 5 and suggest some issues for further investigation.
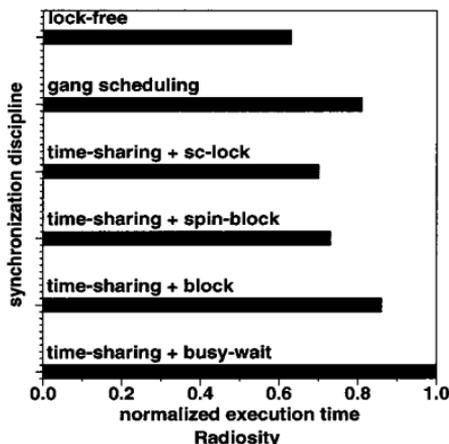
Fig. 10. Performance of lock-free synchronization on a multiprogrammed execution environment executing Radiosity.

Directory-based cache coherence increases the latency and precludes the scalability of synchronization algorithms that use atomic RMW instructions on volatile cached variables. This is attributed to the fact that under high contention between processors, synchronization instructions suffer cache misses on dirty data located in remote memory modules. These cache misses are the most expensive to serve on a ccNUMA multiprocessor. On a highly optimized cache-coherence protocol like that of the Origin2000, a cache miss to remote dirty data is serviced with 5 nonoverlapped network messages.

The implications of directory-based cache coherence do not affect significantly the relative performance of synchronization algorithms that implement the same synchronization semantics. However, in contrast to the common belief that exploiting the memory hierarchy in software is enough to implement scalable synchronization algorithms,[5] our results indicate that software synchronization algorithms will scale only if they use scalable RMW instructions. The scalability of these instructions is not granted on ccNUMA multiprocessors, unless additional hardware is provided to execute these instructions faster. The latency of these instructions can affect significantly the performance of programs with fine-grain parallelism. Although for some of the applications with which we experimented, simple algorithmic modifications or larger problem sizes can eventually alleviate the synchronization bottlenecks,[6] this is not the case in other applications, in particular system software like thread libraries, schedulers and memory managers. System codes synchronize frequently and require scalable synchronization primitives. We therefore conclude that investigating solutions

for faster synchronization on ccNUMA architectures is both worthwhile and important.

Our experiments demonstrated that simple hardware for implementing RMW instructions directly at memory can reduce drastically the overhead of synchronization primitives and, effectively, the latency of synchronization. The type of hardware used in the Origin2000 for this purpose is fairly cheap, compared to full hardware implementations of locks, such as the Queue-on-Lock-Bit[8] or specialized synchronization networks.[16, 43] The at-memory RMW instructions were used in earlier generations of scalable shared-memory multiprocessors without hardware cache coherence, such as the Cray T3D and T3E systems,[42, 44] where they served as the basis for implementing fast synchronization algorithms in software. We demonstrated that these instructions can be used in ccNUMA systems as well, to reduce the latency of practically any software synchronization algorithm. The critical optimization towards leveraging at-memory RMW instructions is to combine them with cached instructions and local spinning, in order to reduce network traffic. Our hybrid implementations of synchronization algorithms appear to be the fastest. The hybrid implementations lie between hardware and software implementations of synchronization algorithms. Although an analytic comparison between software, hardware and intermediate synchronization mechanisms is pending due to some subtle methodological considerations, our evidence from a real system indicate that intermediate schemes that leverage simple hardware and sophisticated software are expected to have the best price/performance ratio.

We investigated thoroughly the potential of lock-free synchronization on ccNUMA multiprocessors. Our results indicate that lock-free synchronization can deliver noteworthy performance gains if the lock-free algorithms can implement exactly the same synchronization semantics as the corresponding algorithms that use spin locks and if the hardware implications do not impede the scalability of the RMW instructions used in the lock-free algorithms. The latter condition is of particular importance, because lock-free algorithms use several expensive RMW instructions as well as remote memory accesses in order to ensure the integrity of concurrent data structures.

Our experimentation with multiprogrammed workloads has revealed several interesting trade-off's between multiprogramming-conscious synchronization algorithms and synchronization-conscious scheduling strategies. Gang scheduling, which is considered to be the scheduling policy of choice for tightly synchronized programs,[20] does not outperform time-sharing when the latter is coupled with a competitive waiting algorithm or scheduler-conscious synchronization. Although we have spent a lot of effort to have an efficient implementation of gang scheduling with our CPU allocator,

gang scheduling has yet to demonstrate the results that would justify its implementation complexity in a real multiprocessor operating system.

Among the synchronization algorithms that we tested with a time-sharing scheduler, scheduler-conscious synchronization appears to be the best, although the performance difference between scheduler-conscious synchronization and competitive waiting is marginal. The primary short-coming of scheduler-conscious synchronization is that a complete imple-mentation requires modifications to the operating system. On the other hand, waiting algorithms can be implemented easily at user-level. We have also shown that lock-free synchronization has robust performance in multi-programmed environments, due to the immunity of lock-free algorithms to inopportune preemptions of threads from the operating system. Our experience with real codes suggests that lock-free synchronization is applicable in several practical implementations.

An interesting question which is left unanswered in this paper is the performance of synchronization algorithms with a space-sharing scheduler. Space-sharing schedulers execute parallel programs in virtually isolated parti-tions of processors. The interesting aspect of space-sharing schedulers is that they may change the actual degree of parallelism in a parallel program at run-time, according to the fluctuation of system load. This is done by preempting and resuming threads that participate in the execution of parallel programs. User-level competitive waiting algorithms are insufficient for space-sharing schedulers. Intuitively, in order to let parallel programs adapt to space-shar-ing, the operating system must provide mechanisms to inform the programs of thread preemptions and let them resume preempted threads. In this aspect, dynamic space sharing can be effectively integrated with scheduler-conscious synchronization, a mechanism that already encompasses a communication channel between parallel programs and the operating system. However, scheduler-conscious synchronization should be extended with techniques that either prevent the preemption of threads that execute inside a critical section (i.e., preemption-safe locking), or raise instantaneously the priority of preempted lock-holding threads, in order to resume them as soon as possible.

As the gap between microprocessor speed and memory access latency increases, synchronization is likely to be a more critical performance bottle-neck for scalable parallel systems. Investigating hardware and software mechanisms for faster synchronization using cost-effectiveness criteria remains an open issue that merits further investigation.

ACKNOWLEDGMENTS

REFERENCES

1. D. Jiang and J. P. Singh, Scaling Application Performance on a Cache-Coherent Multi-processor, *Proc. 26th Int'l. Symp. Computer Architecture (ISCA'99)*, Atlanta, Georgia, pp. 305–316 (May 1999).
2. Y. Solihin, V. Lahm, and J. Torrellas, Scal-Tool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors, *Proc. ACM/IEEE Supercomputing: High Performance Networking and Computing Conf. (SC'99)*, Portland, Oregon (November 1999).
3. T. Anderson, The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, *IEEE Trans. Parallel Distributed Syst.*, **1**(1):6–16 (January 1990).
4. A. Gupta, A. Tucker, and S. Urushibara, The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems (SIGMETRICS'91)*, San Diego, California, pp. 120–132 (June 1991).
5. J. Mellor-Crummey and M. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Trans. Computer Syst.*, **9**(1):21–65 (February 1991).
6. S. Kumar, D. Jiang, R. Chandra, and J. P. Singh, Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance, *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, Atlanta, Georgia, pp. 23–24 (May 1999).
7. D. Nikolopoulos and T. Papatheodorou, A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin2000, *Proc. 13th ACM Int'l. Conf. Supercomputing (ICS'99)*, Rhodes, Greece, pp. 319–328 (June 1999).
8. A. Kägi and J. Goodman, Efficient Synchronization: Let Them Eat QOLB, *Proc. 24th Int'l. Symp. Computer Architecture (ISCA'97)*, Denver, Colorado, pp. 171–181 (June 1997).
9. C. Kuo, J. Carter, and R. Kuramkote, MP-LOCKs: Replacing H/W Synchronization Primitives with Message Passing, *Proc. Fifth Int'l. Symp. High Performance Computer Architecture (HPCA-5)*, Orlando, Florida, pp. 284–288 (January 1999).
10. R. Rajwar, A. Kägi, and J. Goodman, Improving the Throughput of Synchronization by Insertion of Delays, *Proc. Sixth Int'l. Symp. High Performance Computer Architecture (HPCA-6)*, Toulouse, France, pp. 156–165 (January 2000).
11. P. Trancoso and J. Torrellas, The Impact of Speeding Up Critical Sections with Data Prefetching and Forwarding, *Proc. 1996 Int'l. Conf. Parallel Processing (ICPP'96)*, Bloomingdale, Illinois, pp. 79–86 (August 1996).
12. P. Diniz and M. Rinard, Eliminating Synchronization Overhead in Automatically Parallelized Programs Using Dynamic Feedback, *ACM Trans. Comp. Syst.*, **17**(2):89–132 (February 1999).
13. S. Fu and N. Tzeng, A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors, *IEEE Trans. Parallel Distributed Syst.*, **8**(6):628–639 (June 1997).

14. T. Huang, Fast and Fair Mutual Exclusion for Shared Memory Systems, *Proc. 19th IEEE Int'l. Conf. Distributed Computing Systems (ICDCS'99)*, Austin, Texas, pp. 224–231 (1999).

15. B. Lim and A. Agarwal, Reactive Synchronization Algorithms for Multiprocessors, *Proc. Sixth Int'l. Conf. Architectural Support for Progr. Lang. Oper. Syst. (ASPLOS-VI)*, San Jose, California, pp. 25–35 (October 1994).

16. P. Reynolds, C. Williams, and R. Wagner, Isotach Networks, *IEEE Trans. Parallel Distributed Syst.*, **8**(4):337–348 (April 1997).

17. A. Karlin, K. Li, M. Manasse, and S. Owicki, Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor, *Proc. 13th ACM Symp. Oper. System Principles (SOSP'91)*, Pacific Grove, California, pp. 41–55 (October 1991).

18. L. Kontothanassis, R. Wisniewski, and M. Scott, Scheduler-Conscious Synchronization, *ACM Trans. Computer Syst.*, **15**(1):3–40 (February 1997).

19. B. H. Lim and A. Agarwal, Waiting Algorithms for Synchronization in Large-Scale Multiprocessors, *ACM Trans. Computer Syst.*, **11**(3):253–294 (August 1993).

20. D. Feitelson and L. Rudolph, Gang Scheduling Performance Benefits for Fine-grain Synchronization, *J. Parallel Distributed Computing*, **16**(4):306–318 (December 1992).

21. J. Ousterhout, Scheduling Techniques for Concurrent Systems, *Proc. Third Int'l. Conf. Distributed Computing Systems (ICDCS'82)*, Miami, Florida, pp. 22–30 (October 1982).

22. N. Arora, R. Blumofe, and C. Greg-Plaxton, Thread Scheduling for Multiprogrammed Multiprocessors, *Proc. Tenth ACM Symp. Parallel Algorithms and Architectures (SPAA'98)*, Puerto Vallarta, Mexico, pp. 119–129 (June 1998).

23. M. Herlihy, Wait-free Synchronization, *ACM Trans. Progr. Lang. Syst.*, **13**(1):124–149 (January 1991).

24. S. Lumetta and D. Culler, Managing Concurrent Access for Shared-Memory Active Messages, *Proc. of First IEEE/ACM Joint Int'l. Parallel Processing Symp. Symp. Parallel and Distributed Processing (IPPS/SPDP'98)*, Orlando, Florida, pp. 272–276 (April 1998).

25. M. Michael and M. Scott, Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared-Memory Multiprocessors, *J. Parallel Distributed Computing*, **51**(1):1–26 (May 1998).

26. S. Prakash, D. Lee, and T. Johnson, A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap, *IEEE Trans. Computers*, **43**(5):548–559 (May 1994).

27. J. Valois, Lock-Free Data Structures, Ph.D. thesis, Rensselaer Polytechnic Institute, Department of Computer Science (1995).

28. J. Laudon and D. Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server, *Proc. 24th Int'l. Symp. Computer Architecture (ISCA'97)*, Denver, Colorado, pp. 241–251 (June 1997).

29. D. Nikolopoulos and T. Papatheodorou, Fast Synchronization on Scalable Cache-Coherent Multiprocessors using Hybrid Primitives, *Proc. 14th IEEE/ACM Int'l. Parallel and Distributed Processing Symp. (IPDPS'2000)*, Cancun, Mexico, pp. 711–719 (May 2000).

30. D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufman (1998).

31. L. Rudolph and Z. Segall, Dynamic Decentralized Cache Schemes for MIMD Parallel Processors, *Proc. 11th Int'l. Symp. Computer Architecture (ISCA'84)*, Ann Arbor, Michigan, pp. 340–347 (June 1984).

32. M. Herlihy and J. Wing, Axioms for Concurrent Objects, *Proc. 14th ACM Symp. Principles Progr. Lang. (POPL'87)*, Munich, Germany, pp. 13–26 (January 1987).

33. M. Michael and M. Scott, Simple, Fast and Practical Nonblocking and Blocking Concurrent Queue Algorithms, *Proc. 15th ACM Symp. Principles of Distributed Computing (PODC'96)*, Philadelphia, Pennsylvania, pp. 267–276 (1996).

34. B. Marsh, M. Scott, T. LeBlanc, and E. Markatos, First-Class User-Level Threads, *Proc. 13th ACM Symp. Oper. Syst. Principles (SOSP'91)*, Pacific Grove, California, pp. 110–121 (October 1991).

35. X. Martorell, J. Corbalan, D. Nikolopoulos, N. Navarro, E. Polyhchronopoulos, T. Papatheodorou, and J. Labarta, A Tool to Schedule Parallel Applications on Multi-processors: The NANOS CPU Manager, *Proc. Sixth Workshop on Job Scheduling Strategies for Parallel Processing*, in conjuction with *IEEE IPDPS'2000*, Lecture Notes in Computer Science, Vol. 1911, Cancun, Mexico, pp. 87–112 (May 2000).

36. Silicon Graphics Inc. IRIX 6.5 Man Pages. http://techpubs.sgi.com (November 1999).

37. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Int'l. Symp. Computer Architecture (ISCA'95)*, Santa Margherita Ligure, Italy, pp. 24–37 (June 1995).

38. Standard Performance Evaluation Corporation, SPEC CPU95 Documentation. http://www.spec.org (December 1999).

39. X. Martorell, E. Ayguadé, N. Navarro, J. Corbalan, M. Gonzàlez, and J. Labarta, Thread Fork/Join Techniques for Multi-Level Parallelism Exploitation in NUMA Multiprocessors, *Proc. 13th ACM Int'l. Conf. Supercomputing (ICS'99)*, Rhodes, Greece, pp. 294–301 (June 1999).

40. OpenMP Architecture Review Board, OpenMP Fortran Application Programming Interface, Version 1.1 (November 1999).

41. D. Craig, An Integrated Kernel and User-Level Paradigm for Efficient Multiprogramming. Technical Report CSRD No. 1533, University of Illinois at Urbana-Champaign (June 1999).

42. J. Torrellas, A. Tucker, and A. Gupta, Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors, *J. Parallel Distributed Computing*, **24**(2):139–151 (February 1995).

43. R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick, Empirical Evaluation of the Cray T3D: A Compiler Perspective, *Proc. 22nd Int'l. Symp. Computer Architecture (ISCA'95)*, St. Margherita Ligure, Italy, pp. 320–331 (June 1995).

44. S. Scott, Synchronization and Communication in the T3E Multiprocessor, *Proc. Seventh Int'l. Conf. Architectural Support for Progr. Lang. Oper. Syst. (ASPLOS-VII)*, Cambridge, Massachusetts, pp. 26–36 (October 1996).