A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin2000

Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou

High Performance Information Systems Laboratory
Department of Computer Engineering and Informatics
University of Patras
Rio 26500, Patras, Greece
e-mail:{dsn,tsp}@hpclab.ceid.upatras.gr

Abstract

This paper assesses the performance and scalability of several software synchronization algorithms, as well as the interrelationship between synchronization, multiprogramming and parallel job scheduling, on ccNUMA systems. Using the SGI Origin2000, we evaluate synchronization algorithms for spin locks, lock-free concurrent queues, and barriers. We analyze the sensitivity of synchronization algorithms to the hardware implementation of elementary synchronization primitives and investigate in-depth the architectural implications and particularly the tradeoffs between implementing synchronization primitives with cacheable synchronization variables or at-memory. The architectural study enables us to contribute scalable, customized implementations of synchronization algorithms, including a hybrid scheduler-conscious queue lock and a lock-free queue. We also evaluate different combinations of synchronization algorithms, synchronization disciplines that cope with the effects of multiprogramming and different parallel job scheduling strategies, using the Cellular IRIX operating system as a case study.

1 Introduction

Cache Coherent Non Uniform Memory Access (ccNUMA) architectures have recently attracted considerable research and commercial interest, as they present strong advantages in the direction of achieving high performance. At the same time, synchronization is still an intrusive source of bottlenecks in parallel programs for shared memory. The importance of synchronization has motivated a vast amount of research efforts, which contributed several efficient algorithms for tightly-coupled small-scale symmetric multiprocessors (SMPs), as well as scalable algorithms for distributed memory multiprocessors. Whether these solutions are still sufficient for modern ccNUMA systems remains an open and important question.

This paper addresses some significant issues of synchronization on ccNUMA systems. In this direction, we conduct a thorough architectural evaluation of software synchronization algorithms both standalone and in conjunction with spinning and scheduler-conscious disciplines, embedded in synchronization algorithms to cope with the interferences of multiprogramming and the operating system scheduling strategy. More specifically, the main issues addressed in this paper include: (1) The architectural implications of ccNUMA on the scalability of elementary synchronization primitives, as well as the sensitivity of these primitives to their hardware implementation; (2) understanding the behavior and relative performance of synchronization algorithms in terms of hardware performance counts that reflect the interactions between synchronization algorithms, the cache coherence protocol and the memory subsystem; (3) the effectiveness of using specialized hardware that bypasses the caches, to implement scalable synchronization algorithms; (4) how feasible is the implementation of scalable, lock-free and non-blocking synchronization algorithms and how these algorithms perform against standard mutual exclusion algorithms with locks; (5) how different combinations of multiprogramming-conscious synchronization disciplines and parallel job scheduling strategies perform on a contemporary multiprogrammed ccNUMA multiprocessor.

The goal of this work is to provide an in-depth understanding of the ccNUMA architectural impact on software synchronization, as well as the close interrelationship between synchronization, multiprogramming and parallel job scheduling on modern ccNUMA systems. We use the Origin2000 as a case study, since it offers the necessary for our evaluation advanced hardware and software features for scalable synchronization and multiprogramming adaptability of parallel programs. It should be clear however that the conclusions extracted from this study have general applicability in other realizations of the ccNUMA architecture. Aside from the evaluation, this study contributes also some highly efficient, customized implementations of scalable synchronization algorithms on the Origin2000.

After a brief overview of the Origin2000 hardware, we discuss its architectural implications on the scalability of elementary synchronization operations. We then extend the discussion in the context of synchronization algorithms for spin locks, concurrent queues, and barriers. The mutual influence of synchronization, multiprogramming and parallel job scheduling is treated separately. In this context, we examine techniques applied at user-level, or both at the user and kernel levels to achieve scalable synchronization under multiprogramming. Although these techniques are primarily designed to work with time-sharing schedulers, the effectiveness of gang scheduling as a beneficial scheduling strategy for tightly synchronized parallel programs is also studied.

The main body of the evaluation is performed with realistic microbenchmarks executed in dedicated and multiprogrammed environments on a 64-processor Origin2000. The interpretation of the results is based on hardware performance counts, which we extracted on-line during the experiments. The generality of the results is validated using three programs with high synchronization overhead from the SPLASH-2 benchmark suite [25].

ICS '99 Rhodes Greece

To our knowledge, this work is the first to evaluate a broad spectrum of synchronization algorithms and their interrelationship with multiprogramming on an actual ccNUMA platform. Previous studies of synchronization on ccNUMA architectures [6, 10, 13, 17, 23] examined in isolation the scalability of hardware synchronization primitives or the mutual influence of synchronization and multiprogramming, using simulation. A more recent study of synchronization on shared memory multiprocessors [9] uses also the Origin2000 as an evaluation testbed. However, this study focuses mainly on methodological issues for selecting benchmarks to evaluate synchronization and does not examine alternatives such as lock-free and non-blocking synchronization, neither the performance impact of multiprogramming on synchronization algorithms. We examine synchronization from the architectural and the operating system perspective, in order to give valuable practical insights and assist the design of synchronization hardware and algorithms for system architects and end-users of modern and future ccNUMA systems.

The remainder of the paper is organized as follows: Section 2 outlines the Origin2000 architecture and hardware support for synchronization. In Section 3, we overview synchronization algorithms for spin-locks, lock-free queues and barriers and discuss the architectural implications of ccNUMA and the interferences between synchronization algorithms and multiprogramming. Our experimental results with microbenchmarks in dedicated and multiprogrammed environments are presented in Sections 4 and 5 respectively. Section 6 reports results from application benchmarks. We summarize the results in Section 7 and conclude the paper in Section 8.

2 The SGI Origin 2000

2.1 Architectural Overview

The Origin2000 [11] is a ccNUMA multiprocessor introduced by Silicon Graphics Inc. in 1996. The system employs an aggressive memory and communication architecture to achieve high scalability. The building block of Origin2000 is a dual-processor node which consists of 2 MIPS R10000 processors, with 32 Kilobytes of split primary instruction and data caches and up to 4 Megabytes of unified second level cache per processor. Each node contains up to 4 Gigabytes 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 routers are interconnected in a fat hypercube topology. The hub serves as the communication assist for the Origin2000 nodes and implements the cache coherence protocol. The Origin2000 uses a memory-based directory cache coherence protocol with MESI states and a sequentially consistent memory model [11]. Along with aggressive hardware optimizations, the system uses software prefetching and hardware/software support for page migration and replication to reduce the ratio of remote to local memory access time to no more than 3:1 for configurations up to 128 processors. The Origin2000 uses Cellular IRIX, a highly scalable 64-bit operating system with support for multithreading, distributed shared memory and multidisciplinary schedulers for batch, interactive, real-time and parallel processing.

2.2 Support for Synchronization

The Origin2000 provides two hardware mechanisms for interprocessor synchronization. The first mechanism is realized at the processor level and implements a load linked-store conditional (LL-SC) instruction in the cache controller of the MIPS R10000. The instruction is composed of two elementary operations. 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 was completed, otherwise it fails. A successful load linked-store conditional pair 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 be used to implement a wide range of other synchronization instructions, including fetch_and_ϕ, test_and_set and compare_and_swap[1].

The implementation of LL-SC on the R10000 uses a reservation bit per cache line which is set when the load linked is executed [15]. This bit is invalidated before the execution of the store conditional, if the associated cache line is also 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 in order to avoid a livelock situation. The latter could happen if two processors start invalidating each other's cached copy of the synchronization variable without making further progress. The same situation could occur if load linked loaded the variable in exclusive state, in order to avoid a possible coherence cache miss in the store conditional. As a consequence, a successful LL-SC may experience two cache misses, which is likely to happen more frequently if the synchronization variable is heavily contended. In the Origin2000 architecture, many of these coherence misses are satisfied from remote memories, other than the memory of the processor that experiences the miss. Remote misses are serviced with three-party transactions in the Origin2000 coherence protocol [11] and their service has higher latency, depending on the distance between the invlolved processors.

The second hardware synchronization mechanism in the Origin2000 is implemented at the node memory. Specialized hardware is employed to implement atomic operations at-memory. These operations are called fetchops and include atomic loads, stores, fetch_and_and, fetch_and_or, fetch_and_increment, fetch_and_decrement and an exchange with zero. Fetchops operate on 64-byte memory blocks, allocated from a special segment of the node memory which is not replicated in the processor caches. Reads and updates of fetchop memory blocks require a single message in the interconnection network and do not generate coherence traffic. A shortcoming of fetchops is the read latency experienced by a processor that spins on an uncacheable variable, since the read operations issued by the processor go always to a DRAM memory module, which may reside on a remote node. The architects of the Origin2000 have circumvented this by adding a small (one to four-entry) fetchop cache at the memory interface of the hubs, to hold the most recently used fetchop variables. This cache reduces the best-case latency of a fetchop down to approximately the latency of a secondary cache access, when the fetchop is issued to local memory. However, the effective read latency of fetchops is still high and spinning on fetchop variables may generate significant network traffic. A second drawback of fetchops is that they lack a poweful synchronization primitive like compare_and_swap, or atomic exchange of the contents of two memory locations. This is an important limitation, since it precludes the implementation of non-blocking synchronization algorithms. We revisit this issue in Section 3.

---

[1] Compare_and_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 and if so, it stores the new value in the memory location. In both cases the instruction returns an indication if it succeeded or not.

3 Synchronization Algorithms

In this section, we overview the synchronization algorithms which serve our evaluation purposes and discuss the architectural implications of the Origin2000 hardware on the performance of these algorithms. Due to space considerations, the discussion is limited to the fundamental properties and the implementation of the algorithms on the Origin2000. The following three subsections overview algorithms for spin locks, concurrent queues and barriers. The last subsection discusses mechanisms embedded in synchronization algorithms to achieve scalability under multiprogramming.

3.1 Spin Locks

Synchronization with spin locks implies the use of a shared flag which serves as a lock for mutual exclusion. We evaluate three popular flavors of spin locks, the test_and_set lock, the ticket lock and the queue lock.

In the case of the test_and_set lock, an acquiring process attempts to atomically test if the value of the lock is zero and set the value to one. If the process succeeds, it proceeds in the critical section, otherwise it busy-waits until the lock is reset. Upon a lock release, the lock holder resets the lock and the waiting processes race to enter the critical section by issuing test_and_set instructions. On the Origin2000, test_and_set can be implemented either with LL-SC or with the fetchop that atomically exchanges the contents of a memory location with zero.

When implemented with LL-SC, the test_and_set lock introduces hot spots in the memory system at the lock acquire and release phases. Every test_and_set updates the lock (although it does not always modify its value) and with each update invalidations are sent to all processors that cache the lock in shared state. The number of invalidations tends to increase when multiple processors issue the LL-SC sequence simultaneously or race for the critical section upon a lock release. One solution is to poll continuously the lock before issuing a test_and_set [22]. This solution reduces the number of test_and_set's that each processor issues to acquire the lock and therefore the number of invalidations and coherence traffic. However, this solution is inadequate for ccNUMA architectures. In a ccNUMA system, the lock is allocated in the memory module of a single node which contains also the associated directory information. Every processor outside the home node of the lock, retrieves an updated value of the lock from a remote memory location when it experiences a coherence miss on the associated cache line. Therefore, most processors experience potentially expensive cache misses also during the polling phase. A technique to further alleviate this effect is to use backoff, i.e. let the processor sit for a while in an idle loop between successive polls. Although, the implications of the cache coherence protocol impede the scalability of the test_and_set lock, coherence overhead can be eliminated if the lock is allocated as an uncacheable variable. The disadvantage of this solution is the increased latency and network traffic incurred from spinning on the lock variable.

In our implementations of the test_and_set lock with LL-SC and fetchops we used polling and bounded exponential backoff [1] to alleviate hot spotting. The base backoff interval in the implementation with LL-SC is set to be roughly equal to the cost of an uncontended processor read in the secondary cache. In the fetchop implementation, the base interval is set roughly equal to the latency of a fetchop read from local memory. Both approaches attempt to limit the cost payed by unfortunate backoff decisions.

The test_and_set lock is non-deterministic with respect to the order in which processes access the critical section. The ticket lock [16], overcomes this limitation by guaranteeing FIFO service to the processes that acquire the lock. Each process accesses the critical section with a unique ticket number, obtained at the lock acquire phase. A releasing process increments a now_serving variable, which holds the ticket number of the lock holder. Waiting processes spin until their ticket number becomes equal to now_serving. The only atomic synchronization operation needed by the ticket lock is fetch_and_increment, which can be implemented on the Origin2000 either with LL-SC, or with the corresponding fetchop.

The ticket lock has two advantages compared to the test_and_set lock. It guarantees FIFO service for contending processes and reduces the critical path of the lock acquisition phase. However, the ticket lock is also prone to coherence overhead on a ccNUMA architecture. If the implementation of fetch_and_increment uses LL-SC and contention is high, processors are likely to repeat the LL-SC sequence more than once to acquire the lock and each successful store conditional will issue a potentially large number of invalidations. Furthermore, all processes spin on the now_serving variable which becomes a hot spot whenever a releasing process updates the variable. On the other hand, if fetch_and_increment is implemented at-memory, coherence traffic is no longer an issue, but processors will experience high read latencies when spinning on now_serving and network traffic will increase. In the implementation of the ticket lock we used proportional backoff [16] to reduce the effects of contention. The base backoff unit was selected similarly to the case of the test_and_set lock.

The queue lock [1, 16] is based on the idea of maintaining a queue of processes that wait to enter the critical section. Each acquiring process checks if the queue of lock waiters is empty. If so, the process proceeds in the critical section, otherwise it joins the queue and busy-waits on a local flag. A process that exits the critical section visits the head of the queue of lock waiters and resets the local flag of the first waiter. The queue can be maintained in an array or a linked list. The linked list implementation is more space-efficient, however it requires costly atomic operations, namely compare_and_swap and fetch_and_store. We opted for an array implementation of the queue lock on the Origin2000, mainly because it requires only fetch_and_increment, which can be implemented both with LL-SC and fetchops.

The important advantage of the queue lock other than its deterministic FIFO service, is that processes spin on local variables in the lock acquisition phase. This means that at the lock release phase, the lock holder will be involved in a single point-to-point transaction with the first waiter. Therefore, the lock release is performed with a single cache invalidation. Although the lock acquire phase may require an increased number of invalidations under high contention, the queue lock is expected to experience less coherence overhead than both the test_and_set and the ticket lock

Our implementation of the array-based queue lock with fetchops presents an interesting case. The only fetchop variable used in the implementation is the variable that indicates the slot in the array where the next lock waiter is enqueued. All other bookkeeping variables, as well as the queue itself can be maintained on cacheable locations. The implementation results to a hybrid lock, that uses synchronization at memory and point-to-point operations with cacheable shared variables to combine the benefits of both approaches cumulatively. Section 4 reveals that the hybrid queue lock delivers the best performance among the spin locks that we tested on the Origin2000.

3.2 Lock-Free Synchronization with Concurrent Objects

In many practical cases, critical sections in parallel programs consist of a sequence of statements which update a shared data structure, such as a queue or a stack. An idea established in the recent past to implement these operations efficiently, is to replace the lock/unlock pairs that protect the shared structure with an implementation which allows concurrent accesses from many processors, but maintains the state of the data structure coherent during

```
void enqueue (ulong item, struct queue *q) {
 ulong slot = atomic_increment (q->head);
 atomic_store(q->nodes[slot], item);
}

ulong dequeue (struct queue *q) {
 while (true) {
  if (atomic_load(q->head) == atomic_load(q->tail))
   return QUEUE_EMPTY;
  else {
   ulong slot = atomic_load(q->head);
   ulong item = atomic_clear(q->nodes[slot]);
   if (item != NULL)
    atomic_increment(q->head);
   return item;
  }
 }
}
```

Figure 1: Customized lock-free queue implementation with fetchops.

program execution. Intuitively, the benefit of such an approach is to exploit parallelism by overlapping the updates of shared objects and reduce the latency that processes experience when attempting to access a shared object with a lock. An ideal concurrent object satisfies three properties, the non-blocking property, the wait-free property and the linearizability property [5]. These properties guarantee the following: no process trying to update the object will be delayed indefinitely, contending processes will always finish their updates in a limited amount of time and the concurrent object will function as a sequential object where each concurrent update will appear as an atomic uninterruptible operation.

Although an algorithm that satisfies all three properties has not appeared yet in the literature, non-blocking and linearizable algorithms can be implemented on processor architectures that support a universal atomic primitive. Universal primitives resolve an infinite number of contending processors in a non-blocking fashion. Compare_and_swap, LL-SC and atomic exchange of the contents of two memory locations are universal atomic primitives. At least one of the first two is implemented in most commodity microprocessors.

In this paper, we evaluate three non-blocking and linearizable algorithms that implement concurrent FIFO queues. They are attributed to Prakash, Lee and Johnson [21], Valois [24] and Michael and Scott [18] respectively. All three algorithms use compare_and_swap to atomically update the queue and share a common technique for concurrent updates. The algorithms maintain pointers to the two ends of the queue. An enqueuing/dequeuing process reads the pointer to the tail/head of the queue respectively and attempts to modify the contents of the pointed memory location using compare_and_swap. Enqueue and dequeue attempts are repeated until the compare_and_swap succeeds. The differences between the three algorithms lie in the implementation of the queue and the memory management mechanism used to avoid the problem of pointer recycling (also known as the ABA problem), which may occur between the read and the atomic update of a pointer.

Compared to spin locks, concurrent queues have the advantage of allowing multiple enqueue and dequeue operations to be overlapped. However, concurrent objects have also performance limitations. Compare_and_swap is more expensive than other synchronization primitives. Furthermore, simultaneous executions of compare_and_swap on cacheable memory locations incur hot spots. Under high contention, processors may need to issue multiple compare_and_swap instructions in order to complete their updates. Each successful compare_and_swap may therefore motivate a large number of invalidations and remotely satisfied coherence misses. As a result, the latency of concurrent queue operations is significantly higher than the latency of the corresponding sequential operations.

We implemented compare_and_swap with LL-SC on the MIPS R10000. Based on this primitive, we implemented the three non-blocking algorithms for concurrent queues mentioned previously. Unfortunately, compare_and_swap is not implemented at-memory in the Origin2000. In order to assess the performance of lock-free algorithms with uncacheable synchronization variables, we implemented a customized lock-free queue with fetchops, the pseudocode of which is shown in Figure 1. The implementation is an array-based variant of the ticket algorithm, where the queue ends are kept uncached at-memory and used as tickets to access the slots of the queue. The implementation uses the Origin2000 fetchops for atomic load, store, increment and exchange with zero which are not powerful enough to make our lock-free queue non-blocking.

3.3 Barriers

A barrier is a global synchronization point in a parallel program, beyond which no process can proceed unless all other processes have reached the same point. In this paper, we evaluate two software algorithms for barrier synchronization. The first algorithm is a sense reversing centralized barrier [16]. Each processor that arrives at the barrier increments a shared counter atomically and then spins on a sense variable. The last process that arrives at the barrier sets the sense variable and signals the other processes to leave the barrier. The sense variable is toggled after each barrier, to ensure correct execution of successive barriers. On a ccNUMA architecture, cacheable counter and sense flags may become hot spots and prevent the scalability of the centralized barrier, if processors arrive at the barrier roughly at the same time. The counter and sense variables can be allocated in uncacheable locations to avoid hot spots. We implemented either alternative on the Origin2000, using LL-SC and fetchops.

The second algorithm is a two-tree barrier proposed by Mellor-Crummey and Scott [16]. Processor arrivals and wakeups are propagated through two separate trees. A single tree node is assigned to each process and linked to a parent pointer in the arrival tree and child pointers in the wakeup tree. An arriving process waits first for its children to arrive at the barrier and then propagates its arrival to its parent. The process then spins waiting for a wakeup signal from its parent in the wakeup tree. The process that resides at the root of the arrival tree activates the wakeup phase. The branching factors of the two trees are determined empirically to reduce the latency of spinning and optimize the wakeup phase. We used a branching factor of four in the arrival tree and two in the wakeup tree. The tree-barrier does not require any atomic synchronization operations. Spinning and updates on the tree pointers may still increase coherence overhead, however the number of processors that actively share these pointers is low and contention is not expected to be as harmful as in the centralized barrier.

3.4 Synchronization Disciplines under Multiprogramming

Parallel programs with tight synchronization patterns perform often poorly when executed on a multiprogrammed system. The primary reason is that the operating system scheduler decides frequently to preempt synchronizing processes of parallel programs, in order to increase system throughput and ensure fairness. These preemptions are oblivious of any fine-grain interactions among processes in parallel programs. A typical pathological scenario that demonstrates the effect, is the preemption of a process that holds a lock. In that case, the other processes of the same program spin needlessly, waiting for the preempted process to release the lock.

Several solutions were proposed in the past to cope with inopportune process preemptions and the inconsistencies between synchronization algorithms and the operating system scheduling strategy. There are four general solution frameworks: (1) Each process makes an independent decision between busy-waiting and blocking to relinquish the processor and avoid underutilization [7, 12]. This solution is applicable to locks and barriers;(2) a non-blocking synchronization algorithm is used (Section 3.2); (3) the kernel guarantees that a process is not preempted while executing in a critical section [14]; (4) the lock holder polls the status of its peer processes and avoids passing the lock to a preempted process [8]. The latter approach is also called scheduler-conscious synchronization.

Following the first approach, we embedded two multiprogramming-conscious mechanisms in the synchronization algorithms for locks and barriers, namely immediate blocking and competitive spinning. With immediate blocking, an acquiring process blocks as soon as it finds out that the lock is held by another process. Similarly, a process arriving at a barrier blocks if it finds out that there are other processes not arrived at the barrier yet. When the blocked process is eventually rescheduled, it attempts to reacquire the lock or checks the barrier status again.

Immediate blocking may improve processor utilization, since processes do not waste CPU time spinning, but may also cause unnecessary context switches, if processes block unfortunately just before the lock is released or the last process arrives at the barrier. This effect can be harmful for programs with fine-grain synchronization patterns. Competitive spinning [7] makes a compromise, by letting the acquiring process spin for a “while,, before blocking. In this way, utilization is improved without necessarily sacrificing the performance of tightly synchronized programs. The problem with competitive spinning is how to determine the appropriate interval of spinning before blocking. This interval is generally determined empirically. A good approach is to set the spinning interval equal to the overhead of a context switch. This leads to a competitive spinning strategy where the cost paid at each synchronization point is at most twice the cost of a context switch [7]. This is the strategy that we employ in the implementation of competitive spinning in our experiments with spin locks. For barriers, we set the spinning interval to be proportional to the number of processes not arrived at the barrier yet [8].

Non-blocking synchronization algorithms with concurrent objects have an inherent immunity to the undesirable effects of multiprogramming. The reason is that preemption of a process that attempts to update a concurrent object does not prevent other processes from proceeding and updating the object. Previous research [19] has shown that this property is among the strongest arguments that support the use of non-blocking synchronization in multiprogrammed environments.

The last two solution frameworks i.e. preemption-safe locking and scheduler-conscious synchronization require kernel support to establish a communication path between parallel programs and the operating system, in order to let a program request preemption safety or notify the program of a process preemption. These functionalities were not available in Cellular IRIX by the time we performed our evaluation on the Origin2000. An experimental implementation of scheduler-conscious synchronization in IRIX is described in [3]. For the purposes of this paper, we emulated a scheduler-conscious queue lock at user-level, following the methodology of Kontothanassis et.al. [8]. A user-level scheduler emulates the kernel by preempting and resuming processes using the blockproc()/unblockproc() system calls of IRIX. A portion of the system's memory is mapped shared between the user-level scheduler and parallel programs and then used to communicate process preemptions. The process that holds a lock polls the state of the processes waiting in the queue and hands-off the lock to the first non-preempted process. This solution sacrifices the FIFO ordering imposed by the semantics of the queue lock, in order to attain multiprogramming scalability.

The solutions mentioned so far make the assumption that a general-purpose time-sharing scheduler is used from the operating system to execute parallel programs. It has been shown however [23], that the performance of parallel programs with fine-grain synchronization patterns is tightly related to the operating system scheduling strategy. Previous research demonstrated that gang scheduling [4] is more effective for parallel jobs with fine-grain synchronization patterns. Gang scheduling always executes and preempts the processes of a parallel program as a unit, thus giving to the program the illusion of a dedicated system and ensuring that all synchronizing processes are either running or preempted. Gang scheduling is an attractive alternative, since it does not necessitate the use of a specialized synchronization discipline in the program to cope with multiprogramming.

```
barrier();
for (i=0;i<1000000/nprocs;i++) {
   work();
   synchronize();
}
barrier();
```

Figure 2: Synchronization microbenchmark.

Cellular IRIX offers both gang scheduling and time-sharing as alternative options to schedule parallel programs, via the schedctl() system call. We used either option in our evaluation and coupled time-shared programs with multiprogramming–conscious synchronization disciplines.

4 Synchronization Algorithms Performance

In this section, we evaluate the synchronization algorithms presented in Section 3 using microbenchmarks. Section 4.1 presents results from an experiment that evaluates two implementations of a shared counter on the Origin2000, with LL-SC and fetchops. This experiment enables us to assess the performance of the low-level hardware mechanisms for synchronization. In Section 4.2 we present results for spin locks and lock-free queues. These algorithms are evaluated in conjunction, in order to argue directly on the relative performance of lock-free synchronization against synchronization with locks. Section 4.3 evaluates barrier algorithms.

We used the microbenchmark shown in Figure 2 throughout the experiments. Each processor executes its portion of a tight loop with a million iterations, where the processor alternates between synchronizing and working. The synchronize() block may contain any of the synchronization operations under study, i.e. an atomic update of a counter, an enqueue/dequeue operation being either lock-free or protected with locks, or a call to a barrier synchronization primitive. In the work() block each processor executes a loop updating a vector in its local cache. The number of iterations is selected from a uniform distribution with a parameterized mean value provided at runtime. An artificial load imbalance of ±20% is introduced in the working loop. Based on quantitative analyses of synchronization patterns in real applications [9, 10], this benchmark captures realistically the characteristics of tightly synchronized parallel programs.

During the experiments with microbenchmarks, we used the hardware counters of the MIPS R10000 to measure the occurrences of four critical hardware events that reflect coherence activity due to synchronization operations. These events include failed store conditionals, external invalidations or interventions received from the processors and exclusive store/prefetches to shared lines in the secondary cache. When averaged over the number of synchronization operations, failed store conditionals, external invalidations and exclusive store/prefetches provide individually an indication of the average number of processors that contend for the synchronization variable by executing LL-SC simultaneously. These performance counts are therefore a good indication of the degree of contention for the synchronization variable. On the other hand, external interventions reflect the amount of coherence cache misses satisfied

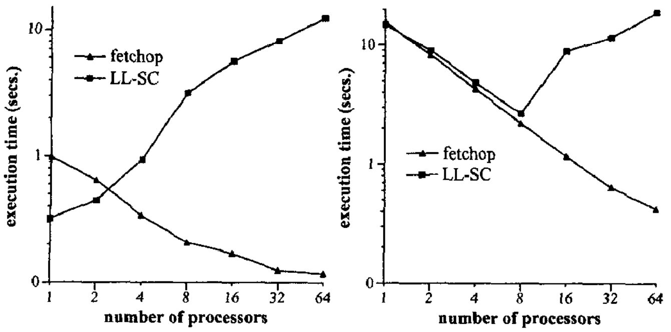

Figure 3: Results from the shared counter microbenchmark. The left chart plots execution time when processors execute empty work loops, while the right chart plots execution time when processors execute on average 5 μs in each work loop.

remotely in the course of execution. In the Origin2000 coherence protocol, an external intervention is sent whenever a processor requests a cache line which is owned exclusively from a processor in a remote node.

We accessed the hardware counters of the R10000 at user level using the /proc interface. The processor has two hardware counters and since we wanted to track four events, we multiplexed two events on each counter and then multiplied the results for each counter by two, to estimate the actual counts. Since all tracked events contribute significantly to execution time, the projections of the counter values are fairly accurate.

All presented measurements in the following sections are averages of five executions and all charts are plotted in logarithmic scale for the sake of readability.

4.1 Shared Counters

In order to evaluate the scalability of the hardware mechanisms for synchronization on the Origin2000, we executed a version of our microbenchmark where each process atomically increments a shared counter at synchronization points. The counter updates are implemented with LL-SC or at-memory with the fetchop_increment instruction.

Figure 3 illustrates the results that demonstrate clearly the advantages of implementing synchronization operations at-memory when the synchronization variable is heavily contested. The at-memory atomic increment exhibits satisfactory scaling up to 64 processors in both versions of the microbenchmark. The flattening of the fetchop curve in the left chart when moving from 32 to 64 processors occurs because the bandwidth of the Origin2000 scales linearly up to 32 processors but flattens when the configuration moves from 32 to 64 processors [11]. The LL-SC implementation of fetch_and_increment does not scale when processors execute empty work loops and scales only up to 8 processors when processors execute non-empty work loops. The left chart in Figure 3 illustrates also the effects of the latency of fetchop accesses. The average latency of the fetchop atomic increment is higher when one or two processors access the shared counter. Accessing the counter from the local cache of the MIPS R10000 is about ten times faster than accessing the counter with a fetchop from local memory. When the benchmark is executed on two processors, synchronization is performed within a single node of the Origin2000. Therefore, coherence transactions do not have to cross the interconnection network and they are performed locally in the hub. The positive effects of caching cease when synchronization crosses the node boundaries. On the other hand, the effect of fetchop latency is mitigated when processors execute non-empty work loops.

Figure 4 plots the R10000 hardware event counts for the shared counter microbenchmark. The results are extracted from executions on 64 processors with non-empty work loops and provide a view of the coherence activity generated from processor contention, when LL-SC is used as an elementary synchronization primitive. At each counter update, 11 processors on average fail to execute the store conditional successfully. Each successful store conditional issues on average 13 invalidations to remote processors and the overall execution incurs 2.5 million coherence cache misses satisfied remotely. The corresponding numbers for the fetchop implementation of the counter are zero and the coherence traffic shown in the chart is attributed solely to cold start misses experienced by the processors at the very first executions of the work loops.

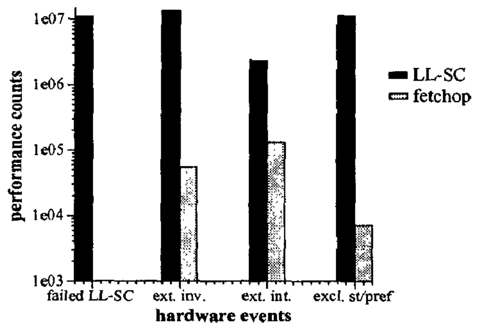

Figure 4: Hardware performance counts of the counter microbenchmark with work loops of 5 μs, executed on 64 processors.

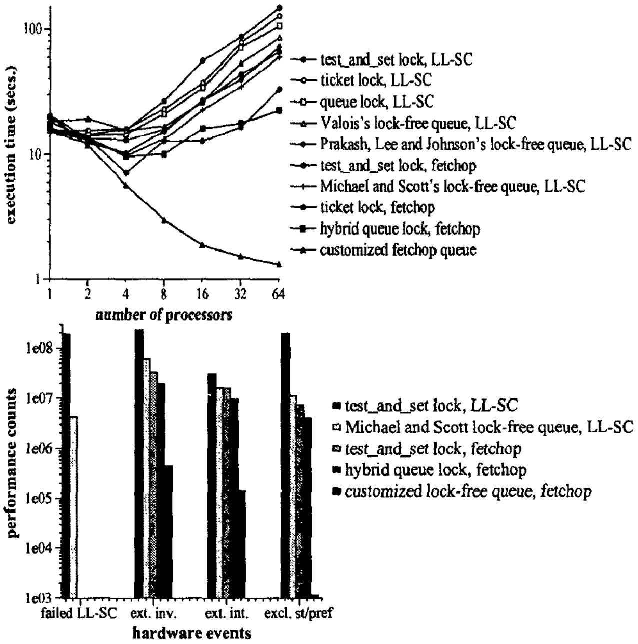

Figure 5: Performance of spin locks and lock-free queues. The top chart plots execution time, while the bottom chart plots the hardware event counts from executions of the microbenchmark on 64 processors, for some illustrative cases. Each process executes on average 5 μs in each work loop.

4.2 FIFO Queues

We use shared FIFO queues to evaluate the performance of spin locks and lock-free synchronization algorithms. We made this choice in order to provide a direct comparison between the two approaches, i.e. synchronization with locks versus lock-free synchronization. The results for the lock-free algorithms are probably the first reported for a ccNUMA architecture.

Figure 5 summarizes the results, including the hardware performance counts from executions of the microbenchmark on 64 processors. We interpret the results in the charts in a top-down manner. The three lock implementations with LL-SC deliver the worst performance. These locks suffer from excessive coherence traffic, an effect which we explored in detail in Section 3.1. The intuition established in Section 3.1 for the relative performance of the three locks is verified from the results in Figure 5. The queue lock outperforms the other locks since it uses only one atomic synchronization operation per lock acquisition and processors spin on locally cached variables. The ticket lock uses also one atomic operation per acquisition but suffers from spinning on a shared location, while the test_and_set lock issues an unpredictable number of atomic operations depending on contention and processors spin also on a shared location. The hardware counts for the test_and_set lock indicate that at each lock acquisition race, on average 90 store conditionals from other processors are issued and fail. The rest of the hardware event counts are not directly interpretable in terms of synchronization operations, since the coherence activity generated from synchronization operations is interleaved with the coherence activity generated from the actual sharing of the queue pointers. A rough indication of the synchronization activity can be extracted by subtracting the counter values of the fetchop implementation of the locks, which reflect the coherence activity generated only from queue sharing. Using this approach, we detect for example that contention due to test_and_set's increases by more than an order of magnitude the number of external invalidations and interventions.

The non-blocking algorithms perform significantly better than locks with cacheable variables, beyond the 4-processor scale. The algorithm of Michael and Scott outperforms the algorithm of Prakash, Lee and Johnson, which in turn outperforms Valois's algorithm. This result was also reported in [18] for bus-based SMPs and it is fairly intuitive, given the complexity of each algorithm and its memory management scheme. The hardware counts for the best performing non-blocking queue indicate that the algorithm reduces the coherence overhead to approximately one fourth of the coherence overhead of the implementations with cacheable locks, although the overall coherence traffic of the non-blocking algorithms remains significant. The reduction of coherence overhead is not translated into a proportional reduction of the execution time of the microbenchmark —which is approximately one half of the execution time of the LL-SC locks microbenchmarks on 64 processors— mainly because of the significantly higher latency of the individual enqueue and dequeue operations of non-blocking algorithms, compared to the latencies of the corresponding sequential operations.

The performance of non-blocking queues based on compare_and_swap is inferior to the performance of fetchop implementations of spin locks. The absence of a universal atomic primitive implemented at-memory in the Origin2000 seems to impede the scalability of non-blocking algorithms. The relative performance of the three locks remains unchanged when the locks are implemented with fetchops. Although the hardware counters cannot track the network activity from fetchops, it appears that the queue lock generates less network traffic compared to the other two locks. Recall from Section 3.1 that our implementation of the queue lock is hybrid. The update of the queue index is done at memory, however processors spin on variables in their caches while waiting for the lock. This combination overcomes the problems of fetchop read latencies and enables the scalability of the queue lock under high contention.

Our customized lock-free queue implemented at-memory delivers the best performance and satisfactory scalability up to 64 processors. All enqueue and dequeue operations are implemented at-memory and coherence activity is generated only from the cold-start misses of the work loops (Figure 5). The read latency of fetchops is not particularly harmful. Although our queue is blocking and does not compare in sophistication with other algorithms for lock-free synchronization, it provides a quite efficient synchronization alternative, suited to the Origin2000 hardware.

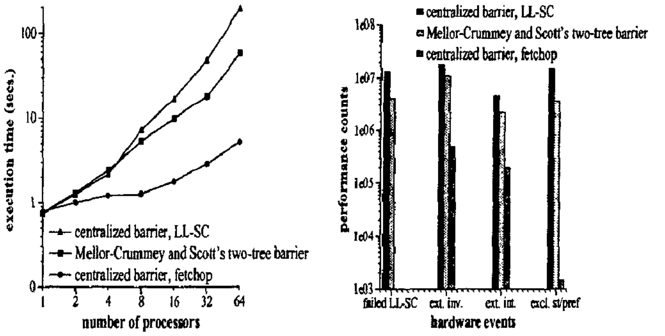

Figure 6: Barriers microbenchmark performance. The left chart plots the performance of different barrier implementations, while the right chart plots the hardware event counts from executions of the barrier microbenchmark on 64 processors. In the microbenchmark, each process executes on average 5 μs in the work loops.

4.3 Barriers

Figure 6 depicts the performance of two implementations of the centralized barrier and the two-tree barrier of Mellor-Crummey and Scott. The results demonstrate that the centralized barrier implemented with fetchops delivers the best performance. The LL-SC implementation of the centralized barrier performs poorly due to its excessive coherence overhead. Interestingly, although the two-tree barrier appears to be scalable, it performs radically worse than the centralized fetchop barrier. The hardware performance counts (right chart) indicate that although the two-tree barrier does not issue any synchronization operations, it still incurs significant coherence overhead and network traffic, despite the fact that the variables used in the barrier implementation are actively shared only by pairs or quads of processors. It seems that there is no need to implement a highly sophisticated tree barrier in the Origin2000, since the centralized fetchop barrier performs very well, at least up to a 64-processor scale.

5 Performance under Multiprogramming

In order to assess the impact of multiprogramming and parallel job scheduling on synchronization in ccNUMA systems, we evaluated five different synchronization disciplines, busy-wait, immediate blocking, competitive spinning, scheduler-conscious synchronization and non-blocking synchronization, in conjunction with two scheduling strategies of Cellular IRIX, gang scheduling and time sharing. Details on the IRIX schedulers can be found in [2, 3]. For the purposes of this evaluation, we used multiprogrammed workloads, with multiple copies of our FIFO queues and barrier microbenchmarks, running simultaneously on all 64 processors of the Origin2000 on which we experimented. The exception is the scheduler conscious queue lock where the numbers are extracted from executions on 63 processors, since one processor emulates the kernel scheduler. This minor interference does not affect significantly our quantitative comparisons.

Figure 7 illustrates the results from the experiments with multiprogrammed workloads of the queues microbenchmark. The primary outcome of the results is that for all synchronization algorithms, gang scheduling performs significantly worse than time sharing. The performance gap between the two scheduling strategies tends to be wider for the algorithms that use LL-SC. The hardware performance counts of the multiprogrammed executions (not

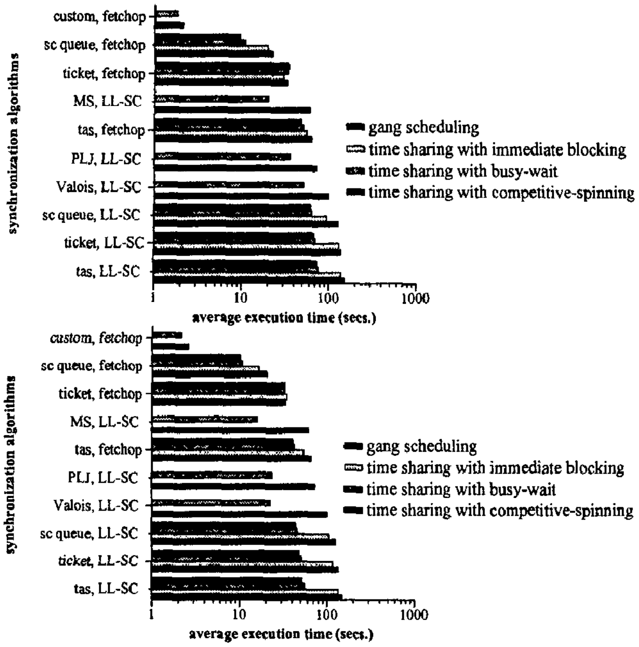

Figure 7: Average execution time of the queue microbenchmarks under two-way (top) and three-way (bottom) multiprogramming on 64 processors with four combinations of synchronization disciplines and IRIX scheduling algorithms. The queue lock is implemented with a scheduler-conscious discipline. Competitive spinning and immediate blocking are applicable only to spin locks. Processes execute on average 50 μs in the work loops.

shown) indicate that when the processes of synchronizing programs are gang scheduled, contention and coherence traffic are exacerbated. We are not in a position to attribute this effect to the implementation of gang scheduling in the IRIX kernel since we did not have access rights to the IRIX source code. We detected from the experiments however that IRIX uses a flat model of gang scheduling, with no dynamic control of the number of processes in each program that are coscheduled under multiprogramming. The time-sharing scheduler alleviates contention by reducing the number of processes that contend actively for a lock or a queue. This effect is somehow artifactual, since the time-sharing scheduler does not take any special provision for synchronization in parallel programs. It indicates however, that dynamic control of the number of each parallel job's processes at runtime can be beneficial for fine-grain synchronization.

The second observation is that with time sharing, non-blocking algorithms tend to perform better than multiprogramming-conscious spin locks, even when the locks are implemented with fetchops. The result demonstrates the robustness of non-blocking algorithms in multiprogrammed environments. The same argument holds for our customized lock-free queue. Although our queue is blocking, it appears that this algorithmic limitation is not particularly harmful. The exception to the above rule is our hybrid scheduler-conscious queue lock, which retains its scalability and adapts effectively to the operating system scheduler through the preemption-safe lock passing mechanism.

The final observation concerns the relative performance of synchronization disciplines with a time-sharing scheduler. Immediate blocking performs worse than the other synchronization disciplines in all cases. The reason is that immediate blocking leads to an excessive number of unnecessary context switches. As long as the latency of context switches in modern operating systems remains high, immediate blocking can be beneficial only to parallel programs with fairly coarse-grain synchronization patterns and under heavy multiprogramming load. However, even in these cases, the effectiveness of immediate blocking was not verified in our experiments with spin locks. Competitive spinning on the other hand had always a positive, but generally marginal effect.

Figure 8 illustrates the results from multiprogrammed experiments with the barrier microbenchmark. The primary conclusion extracted from the results is that gang scheduling performs better for tightly synchronized programs with barriers. Simultaneous scheduling of all processes of each microbenchmark tends to handle well the artificial load imbalances that we introduced in the microbenchmark. However, results from experiments with increased load imbalance and higher degrees of multiprogramming, indicated that gang scheduling was less effective and evidently inferior to time-sharing, when the barrier algorithms were coupled with immediate blocking or competitive spinning. This is illustrated by the rightmost chart in Figure 8. Competitive spinning was beneficial in general, both with tightly and loosely synchronized barrier microbenchmarks.

6 Results from Application Benchmarks

We selected three applications from the SPLASH-2 benchmark suite [25], Cholesky, Radiosity and LU, to assess the performance of synchronization algorithms under realistic conditions. All three applications spend on average around 25% of their execution times at synchronization points, assuming a perfect memory system [25].

For the purposes of the evaluation, we applied minor modifications in the synchronization code of each application. In LU, we modified the code that implements the barriers. In Cholesky and Radiosity, we modified the implementation of the task queues used for better data locality and load balancing. In the experiments, we selected problem sizes that ensured the scalability of the programs, at least up to 32 processors.

Due to space considerations, we present only the results from multiprogrammed executions of the benchmarks. The results from executions in dedicated mode can be found in the expanded version of this paper [20]. The conclusion drawn from these results is that they agree qualitatively with the results from our microbenchmarks, although the effects of using more scalable synchronization algorithms become noticeable (i.e. lead to improvements of more than 2% in execution time) beyond the 32-processor scale.

Figure 9 illustrates the results from multiprogrammed executions of the SPLASH-2 benchmarks. For Cholesky and Radiosity the comparative results agree remarkably with the results from our multiprogrammed experiments with microbenchmarks. The relative performance of combinations of synchronization disciplines and scheduling strategies in these benchmarks remains unchanged compared to their performance with microbenchmarks. The notable exception is LU, where due to significant load imbalance, competitive spinning combined with a time-share scheduler delivers better performance than gang scheduling.

7 Summary of Results

Our experimental results demonstrate the scalability advantages of implementing synchronization primitives at-memory for heavily contested synchronization variables. This observation was validated for all synchronization constructs that we evaluated in this paper. The main drawback of implementing synchronization operations at-memory i.e. the read latency experienced from processors that spin on uncached variables allocated on remote memory modules, can be surmounted with hybrid implementations of synchronization algorithms that combine spinning on cacheable variables with atomic updates on uncacheable variables. We demonstrated the effectiveness of this solution with our hybrid queue lock.

On the other hand, the absence of a universal atomic primitive precluded the implementation of non-blocking algorithms with

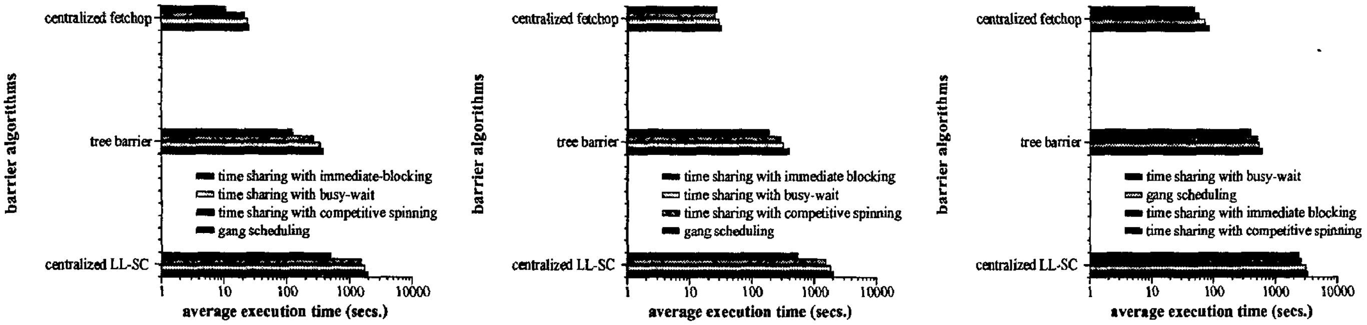

Figure 8: Performance of combinations of barrier synchronization disciplines and IRIX scheduling strategies, under two-way (left) and three-way (middle) multiprogramming on 64 processors. Processes execute on average 50 μs in the work loops. The rightmost chart is extracted from an experiment with a modified microbenchmark with ±50% of load imbalance, under two-way multiprogramming.

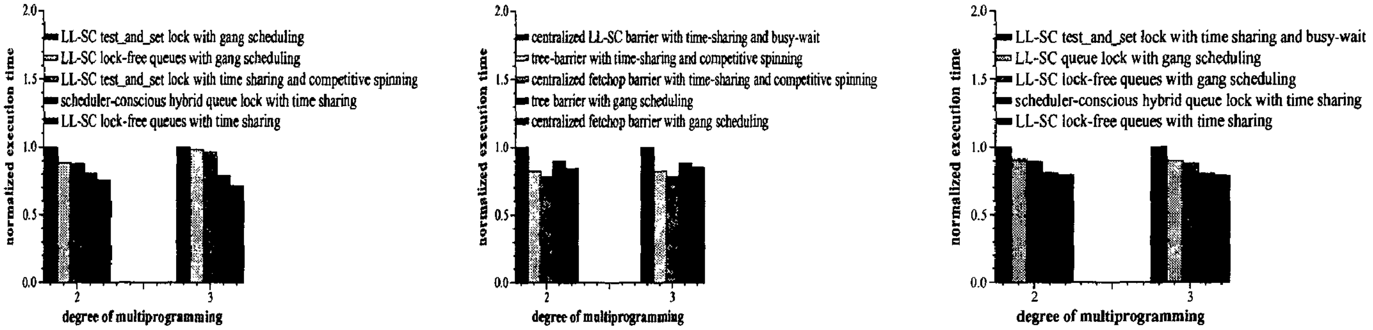

Figure 9: Normalized average execution times of three SPLASH-2 applications, from multiprogrammed experiments on 64 processors. The charts from left to right correspond to multiprogrammed executions of Cholesky, LU and Radiosity. Normalization is performed against the average execution time of the worst performing combination of synchronization discipline and operating system scheduling strategy.

compare_and_swap at-memory. Although we cannot argue for the hardware complexity of adding a universal primitive in the synchronization operations performed at-memory in the Origin2000, we believe that such an atomic primitive is necessary for implementing efficient non-blocking synchronization algorithms for moderate and large-scale multiprocessors. This observation can serve as a guideline for the implementation of synchronization primitives in future hardware generations.

From a programmer's perspective, our results support lock-free synchronization, the efficiency of which is mainly demonstrated by our customized concurrent queue and by the fact that non-blocking algorithms are superior to mutual exclusion algorithms with locks when implemented with the same hardware synchronization primitive. We recommend the use of non-blocking algorithms for the implementation of simple shared objects like queues, stacks and heaps in parallel programs.

In our experiments with multiprogrammed workloads, we observed tradeoffs between selecting a multiprogramming-conscious synchronization discipline and a synchronization–conscious scheduling strategy. Applications with frequent locking perform better in a time-sharing environment, when the synchronization mechanisms are equipped with a scheduling discipline that copes with the preemptions of processes from the operating system. Our results for gang scheduling are to some extent in contrast with previous research results that favored gang scheduling for fine-grain synchronization. We find that in the Origin2000, contention and coherence overhead outweigh the benefits of gang scheduling, although this could be attributed also to the implementation of gang scheduling in the IRIX kernel[2]. On the other hand, we find that the effects of inopportune process preemptions under time-sharing can be dealt with, using either scheduler-conscious synchronization or competitive spinning. Our hybrid scheduler-conscious queue lock has proven to be very robust under multiprogramming. The use of competitive spinning instead of busy-wait was also beneficial in all our experiments. Non-blocking algorithms finally, exhibited a remarkable immunity to multiprogramming effects, a result which was fairly expected. We believe that non-blocking synchronization should be the strategy of choice for achieving multiprogramming scalability of parallel programs on ccNUMA systems, primarily due to its inherent simplicity. Competitive spinning is largely parametric and application-dependent, while scheduler-conscious synchronization requires modest modifications to the operating system kernel, which are not feasible for non-privileged end-users.

With respect to barrier synchronization, we find that gang scheduling tends to perform better with multiprogrammed workloads of programs with fine-grain load imbalances. For programs with more significant load imbalances, gang scheduling leads to poorer utilization. In that case a blocking or competitive spinning strategy with a time-sharing scheduler is more appropriate.

8 Conclusions

This paper contributed a quantitative study of several synchronization algorithms and their interrelationship with multiprogramming and parallel job scheduling on an actual ccNUMA platform. The Origin2000 gave us the opportunity to evaluate these algorithms with advanced hardware and a well-tuned proprietary operating system. Besides several comparative results that were presented thoroughly in Sections 4 through 6, this study brought out several important issues, related to the scalability of synchronization constructs on ccNUMA architectures. We summarize these issues be-

---

[2] The use of gang scheduling in newer versions of IRIX is obsolete and the default environment for multiprogrammed execution of parallel applications uses the time-sharing scheduler and dynamic thread control [3].

low and point out those of them that merit further investigation.

We experienced a strong architectural impact on the performance of synchronization constructs. Although this was expected, our results demonstrated that synchronization algorithms are very sensitive to the hardware implementation of the elementary synchronization primitives on which these algorithms are based. In certain cases, this sensitivity plays a more catalytic role in the performance of synchronization algorithms than their actual algorithmic properties. This effect was demonstrated clearly with the relative performance of non-blocking synchronization algorithms and spin locks, when the latter were implemented at-memory.

We found that detailed knowledge of the ccNUMA architecture and particularly the cache coherence protocol and the memory subsystem is necessary to evaluate synchronization algorithms in the proper context. This detailed knowledge enabled us to contribute efficient customized implementations of synchronization algorithms for the Origin2000 such as the hybrid queue lock and the simple lock-free FIFO queue. The implementation of a complete set of hybrid synchronization algorithms with high scalability on large-scale ccNUMA systems is within our future plans.

Our study on the mutual influence of synchronization and parallel job scheduling reveals that this issue needs further investigation. The experiments with Cellular IRIX, a fairly customized operating system, show that time sharing is not as harmful for synchronization as previous studies demonstrated. Both scheduler-conscious synchronization disciplines and non-blocking synchronization are able to cope with the undesirable interferences of a time-sharing scheduler, while at the same time, time sharing has the ability to alleviate the effect of hot spotting at synchronization points. Gang scheduling did not exhibit the expected performance benefits. A further investigation of this subject in conjunction with more elaborate multiprocessor scheduling strategies with dynamic process control is also planned as future work.

Acknowledgements

We would like to thank Constantine Polychronopoulos and David Craig for motivating this work and providing useful advice. We are also grateful to Oriol Riu and George Tsolis for their invaluable technical assistance.

References

[1] T. Anderson, *Operating System Support for High Performance Multiprocessing*, PhD Thesis, Univ. of Washington at Seattle, 1991.

[2] J. Barton and N. Bitar, *A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX*, Proc. First Workshop on Job Scheduling Strategies for Parallel Processing, 1995.

[3] D. Craig, *An Integrated Kernel-level and User-level Paradigm for Efficient Multiprogramming*, MSc Thesis, Univ. of Illinois at Urbana-Champaign, 1999.

[4] D. Feitelson and L. Rudolph, *Gang Scheduling Performance Benefits for Fine-grain Synchronization*, Journal of Parallel and Distributed Computing, 16(4), pp. 306–318, 1992.

[5] M. Herlihy, *Wait-free Synchronization*, ACM Trans. on Programming Languages and Systems, 11(1), pp. 124–149, 1991.

[6] A. Kägi, D. Burger and J. Goodman, *Efficient Synchronization: Let Them Eat QOLB*, Proc. 24th Int'l Symp. on Computer Architecture, pp. 170–180, 1997.

[7] A. Karlin, K. Li, M. Manasse and S. Owicki, *Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor*, Proc. 13th ACM Symp. on Operating Systems Principles, pp. 41–55, 1991.

[8] L. Kontothanassis, R. Wisniewski and M. Scott, *Scheduler Conscious Synchronization*, ACM Trans. on Computer Systems, 15(1), 1997.

[9] S. Kumar, D. Jiang, R. Chandra and J. P. Singh, *Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance*, to appear in ACM SIGMETRICS'99 Conf., 1999.

[10] C. Kuo, J. Carter and R. Kuramkote, *MP-LOCKs: Replacing H/W Synchronization Primitives with Message Passing*, Proc. of the Fifth Int'l Symp. on High Performance Computer Architecture, 1999.

[11] J. Laudon and D. Lenoski, *The SGI Origin: A ccNUMA Highly Scalable Server*, Proc. 24th Int'l Symp. on Computer Architecture, pp. 241–251, 1997.

[12] B. Lim and A. Agarwal, *Waiting Algorithms for Synchronization in Large-Scale Multiprocessors*, ACM Trans. on Computer Systems, 11(3), pp. 253–294, 1993.

[13] B. Lim and A. Agarwal, *Reactive Synchronization Algorithms for Multiprocessors*, Proc. of ASPLOS-VI, pp. 25–35, 1994.

[14] B. Marsh, M. Scott, T. LeBlanc and E. Markatos, *First-Class User Level Threads*, Proc. 13th ACM Symp. on Operating System Principles, pp. 110–121, 1991.

[15] MIPS Technologies Inc., *MIPS R10000 Microprocessor User's Manual*, Version 2.0, 1997.

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

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

[18] M. Michael and M. Scott, *Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*, Proc. 15th ACM Symp. on Principles of Distributed Computing, 1996.

[19] M. Michael and M. Scott, *Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors*, Proc. 11th Int'l Parallel Processing Symp., 1997.

[20] D. Nikolopoulos and T. Papatheodorou, *A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on the SGI Origin2000*, Technical Report HPCISL-010998, Univ. of Patras, 1998.

[21] S. Prakash, D. Lee and T. Johnson, *A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap*, IEEE Trans. on Computers, 43(5), pp. 548–559, 1994.

[22] Z. Segall and L. Rudolph, *Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor*, Proc. 11th Int'l Symp. on Computer Architecture, pp. 340–347, 1984.

[23] A. Tucker, A. Gupta, and S. Urushibara, *The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications*, Proc. ACM SIGMETRICS'91 Conf., Revised Version, 1995.

[24] J. Valois, *Lock-Free Data Structures*, PhD Dissertation, Rensselaer Polytechnic Institute, 1995.

[25] 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. on Computer Architecture, pp. 24–36, 1995.