# MECHANISMS FOR EFFICIENT SHARED-MEMORY, LOCK-BASED SYNCHRONIZATION

BY

ALAIN KÄGI

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the University of Wisconsin—Madison

1999

# Abstract

Efficient locking synchronization primitives are essential for achieving high performance in fine-grain, shared-memory parallel programs. One function of locking primitives is to enable exclusive access to shared data and critical sections of code. In this dissertation, I make the following six contributions. (1) I propose a framework, the *synchronization period*, in which to reason about the inefficiencies of locking primitives. (2) I identify four previously proposed locking mechanisms (*local spinning*, *queue-based locking*, *collocation*, and *synchronous prefetch*) and uses them to classify existing locking primitives according to which of these mechanisms they incorporate. (3) With detailed simulations, I show the extent to which these four mechanisms can improve the performance of shared-memory programs. I evaluate the space of these mechanisms using sixteen synchronization constructs, which are formed from six base types of locks (test&set, test&test&set, MCS, LH, M, and QOLB). I show that large performance gains (speedups of more than 1.5 for three of six benchmarks) can be achieved if at least three locking mechanisms are used simultaneously. I find that QOLB, which is the first primitive to incorporate all four mechanisms, outperforms all other primitives in all cases. I also find that test&test&set with the collocation mechanism and the exponential back-off policy is able to achieve, in most cases, a performance comparable to QOLB without collocation. (4) I identify six hardware mecha-

nisms required to support each of the four locking mechanisms (*naming*, *protocol processing*, *synchronous cache-to-cache transfer*, *place holder allocation*, *non-blocking instructions*, and *association of lock and data*). For each of these hardware mechanisms I discuss several implementation alternatives. (5) I show that is it possible to implement on today's hardware an efficient synchronization primitive that incorporates the four locking mechanisms. To this end, I describe and evaluate SOFTQOLB, a software implementation of QOLB on a departmental cluster of unmodified workstations. (6) I propose a new locking primitive, called VAQUM, that has, with the programmer's cooperation, the potential to outperform existing primitives. I describe and evaluate CLEAN, a distributed shared-memory system that implements VAQUM.

*A ma mère,*
*à mon père*

# Acknowledgments

I have benefited from the help and support of a great many people while completing this dissertation. I would like to take this opportunity to thank them.

First, I would to like thank my advisor Jim Goodman for years of support and guidance, and for all that he has taught me, both as an architect and as a researcher. I would also like to thank Mary Vernon, David Wood, Guri Sohi, and Parmesh Ramanathan for their time and effort spent reviewing and providing great advice that helped me improve the presentation of this work. Mark Hill, Jim Smith, and Michael Ferris also provided invaluable guidance and encouragement.

I am truly grateful to Scott Breach, Doug Burger, Babak Falsafi, and T. N. Vijaykumar, who were my closest friends during these years spent in Madison and made my stay in graduate school more enjoyable. I am also indebted to Todd Austin, Andy Glew, Stefanos Kaxiras, Alvy Lebeck, Andreas Moshovos, Subbarao Palacharla, Dionisios Pnevmatikatos, Ravi Rajwar, Steve Reinhardt, and Ioannis Schoinas. Finally, I owe my deepest gratitude to Tia Newhall and Martha Townsend, who provided me with a shelter and helped me maintain a semblance of sanity in the final stretch.

Most of all, I would like to thank my parents Résí and René Kägi, and my sister Florence Rikly for their love and support.

This work was supported in part by NSF Grants CCR-9207971 and CCR-9509589, funding from the Apple Computer Advanced Technology Group, an unrestricted grant from the Intel Research Council, and equipment donations from Sun Microsystems. The Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618, with matching funding from the University of Wisconsin Graduate School.

# Contents

# List of figures

# List of tables

# Chapter 1

# Introduction

At any given time, the available technology limits the performance achievable by a single processor. However, a parallel system composed of multiple processors has the potential to overcome this limitation. Ideally, these processors are all performing useful computation. In other words, processors in conjunction are only executing instructions that a single-processor system would execute. No processor is duplicating work and no processor is idle. Unfortunately, parallel systems seldom realize this ideal execution. One of many factors contributing to inefficiencies is the focus of this thesis, namely, inefficiencies in coordinating and synchronizing activities performed on the processors. Other sources of inefficiencies include, for instance, load imbalance when a node finishes work ahead of the others, inter-processor communication delays, and scheduling inefficiencies.

Achieving ideal performance for any program on a parallel system is almost certainly an utopian goal. However, inefficiencies need not be completely overcome for such a system to be cost-competitive with single-processor systems [WH95].

Thus, even if the perfect execution is not achievable, the goal is to wrest the best performance possible from a parallel architecture.

## 1.1 Problem statement

This thesis considers the inefficiencies introduced by synchronization activities. Specifically, the thesis addresses the questions of how to reduce or even to eliminate said inefficiencies, as well as how to implement efficient synchronization support on current and future systems. This work focuses on lock-based synchronization in the context of shared-memory multiprocessors running parallel shared-memory programs.

A *multiprocessor* is a computing system composed of multiple processors and memory modules connected together through an interconnection network (see illustration in Figure 1.1). Each node of the network may attach either a processor, or a memory module, or both. This network permits the different nodes to exchange data by sending requests and receiving responses. The purpose of this configuration is to exceed the computing power achievable by a single-processor system; this thesis does not consider the use of multiple processors for fault tolerance achieved through redundancy.

Multiprocessors typically support either the message passing or the shared-memory programming paradigm (sometimes both). In the traditional message passing model, processors communicate with each other by accessing the interconnection network directly to send and receive messages. This paradigm allows the programmer to optimize data movement among the nodes in the system and

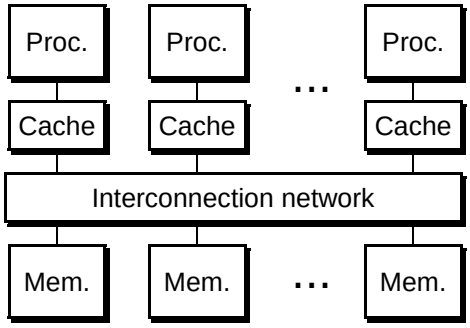



**Figure 1.1 Shared-memory multiprocessor.** A minimal shared-memory multiprocessor configuration consists of a collection of processors (with caches) and memory modules connected together by an interconnection network.

to use the characteristics of the network optimally (e.g., optimize for delivery speed or optimize for throughput). However, the message passing model forces programmers to specify explicitly any communication required among the nodes in the system. Such specification may be challenging to implement both efficiently and correctly, and may, ultimately, restrict the parallelization strategy.

In contrast, for programmers, the *shared-memory* model is an attractive programming paradigm for multiprocessors. This model provides programmers with an intuitively appealing programming model as well as affording reasonably efficient implementations. First, a programmer's intuition developed on single-processor systems still holds on shared-memory multiprocessors, since a load from any node of a (shared-) memory location still returns the value last written in that location. This model implicitly supports communication among processors in the multiprocessor through conventional load and store instructions. The underlying memory system will satisfy a processor's request to read or write a remote memory location by sending and receiving messages to the appropriate memory

module on the processor's behalf. Second, implementations of the shared-memory model provide efficiency through replication of data in local caches associated with each processor in the system. In other words, to remove the necessity of remote communication on every access, the memory system temporarily keeps copies of values locally.

A shared-memory multiprocessor executes multiple threads of control; typically at least one on each processor. Without loss of generality, I will assume that all threads of control execute the same program, each thread typically working on a different part of the data set. This execution model is known as the *single program, multiple data streams* (SPMD) paradigm. A related execution model appropriate for multiprocessors is the *multiple instruction streams, multiple data streams* (MIMD) paradigm, in which each processor executes different instructions operating on different data. However, MIMD is not inherently more powerful than SPMD. Indeed, it is easy to simulate behavior of the MIMD model with an SPMD program: each processor executes different instructions by running in a distinct section of the program.

In the SPMD execution model, each processor of the shared-memory multiprocessor executes at least one instance of the parallel program. I will refer to such an instance as a *process*. It is conceivable to execute many processes multiplexed on each processor of the system. Indeed, this possibility has existed for decades on uniprocessors and is the reason that shared-memory multiprocessors seem "natural." The goal of running more than one process per processor is to increase the utilization of the processors in the system. Multiplexing attempts to enhance pro-

cessor utilization by switching to another process whenever the process currently executing is idle waiting for an event unlikely to occur immediately. Additional processes can worsen synchronization inefficiencies, since contention to synchronization resources will increase. This thesis does not study the impact of multiplexing on the inefficiencies and performance of synchronization activities; therefore I will assume that a single process executes per processor. However, the solutions to reduce lock contention studied in this thesis are likely to benefit these systems also.

The processes of a parallel program running on a shared-memory multiprocessor communicate by writing new values in memory that other processes later read. The correctness of many parallel programs depends on the ability of a process to observe new memory values at the "right moment." A process might start a sequence of changes that leaves a data structure temporarily inconsistent; as a result other processes should not attempt to read the data structure until the sequence has completed. A good example of this concept is the transfer of a sum of money from a bank account (say Alfred's) to another (Béatrice's). This transaction requires two steps: (1) debit Alfred's account by the sum of money, and (2) credit Béatrice's account by the same amount. A snapshot of all account balances taken after step (1) but before step (2) is inconsistent with the total amount of money stored at the bank: it is short by the amount of the transfer. In contrast, a snapshot taken either before or after the execution of the transaction produces a consistent view of the bank accounts. Accordingly, for correctness, some sequence

of modifications or “transaction” must take place atomically: these modifications must appear completely performed or not yet performed at all.

A popular method to ensure that a set of modifications appears atomic is to couple them with a *lock*. Once a process *acquires* a lock, no other processes will be able to do so until the current lock holder *releases* it. By convention, then, processes wishing to perform an atomic transaction must acquire a corresponding lock first, which, once granted, guarantees its current owner that no other processes will be attempting to access locations touched by the transaction. When the lock owner has finished its set of accesses, it grants access to these locations to other processes by releasing the lock.

The instructions implementing lock acquisition and release are commonly referred to as synchronization or locking primitives. These primitives may correspond to a single instruction or to a sequence of instructions implementing a more elaborate algorithm. For instance, if the only atomic memory operations available are load and store instructions, implementing a locking primitive will require a non-trivial algorithm such Lamport’s bakery algorithm [Lam74]. On the other hand, a processor may implement a complete synchronization solution in its instruction set requiring a single instruction to release or to acquire a lock.

Another popular method to coordinate activities in parallel programs are barriers. Although barriers are important to efficient shared-memory programs, they are well understood and many efficient implementations have been proposed or built [AC89, GLR83, KS93, LAD+92, PBG+85, Sco96, UIT94]. In this study, I focus on providing more efficient mutual exclusion through better locks.

It is important that an implementation minimize the delays when accessing either an uncontested or a contested lock. On one hand, a program should both be able to acquire and to release an uncontested lock quickly in order to maximize the serial performance of an individual processor. On the other hand, since the access to a lock is by definition serialized among processors, large inefficiencies when accessing a contested lock may degrade both performance and scalability. To maximize both the performance of parallel programs that use locking and the potential to scale to larger numbers of processors, a lock implementation must also minimize the delays associated with the transfer among caches of values accessed by an atomic transaction.

In the view of the importance of efficient synchronization for parallel programs that use locking on shared-memory multiprocessors, the goals of this thesis are as follows:

- Identify the sources of inefficiencies in lock synchronization
- Compare the performance of locking primitives proposed to-date
- Propose and evaluate improved primitives
- Develop implementation alternatives of the new primitives

## 1.2 Thesis contributions

This thesis makes six distinct contributions. First, it proposes a framework, the *synchronization period*, in which to reason about the inefficiencies of locking primitives. Second, it identifies four previously proposed locking mechanisms (*local spinning*, *queue-based locking*, *collocation*, and *synchronized prefetch*) and

uses them to classify existing locking primitives according to which of these mechanisms they incorporate. Third, it presents the first quantitative evaluation comparing the performance benefits of each individual locking mechanism. It also presents the first quantitative evaluation of the locking primitive called QOLB [GVW89] with a microbenchmark and six shared-memory parallel applications comparing its performance with sixteen previously proposed locking constructs including test&set, test&test&set [RS84], MCS [MCS91a, MCS91b], and the reactive synchronization algorithm [LA94]. Fourth, it discusses practical issues in implementing each of the identified locking mechanisms on current and future shared-memory multiprocessors. Fifth, I show that it is feasible to implement on today's hardware an efficient synchronization primitive that incorporates the four proposed locking mechanisms. Finally, it proposes a new locking primitive, called VAQUM, that has, with the programmer's cooperation, the potential to outperform existing primitives. In the next five subsections, I discuss each contribution in more detail. A review of related work, and how it contrasts with these contributions, appears in Section 1.3.

### 1.2.1 Synchronization period

To understand where the opportunities for optimization lie, I divide the time associated with a complete locking period into three phases: *transfer*, *load/compute*, and *release* (see illustration in Figure 1.2). Together, these phases form a *synchronization period*, which determines an upper bound on the global through-

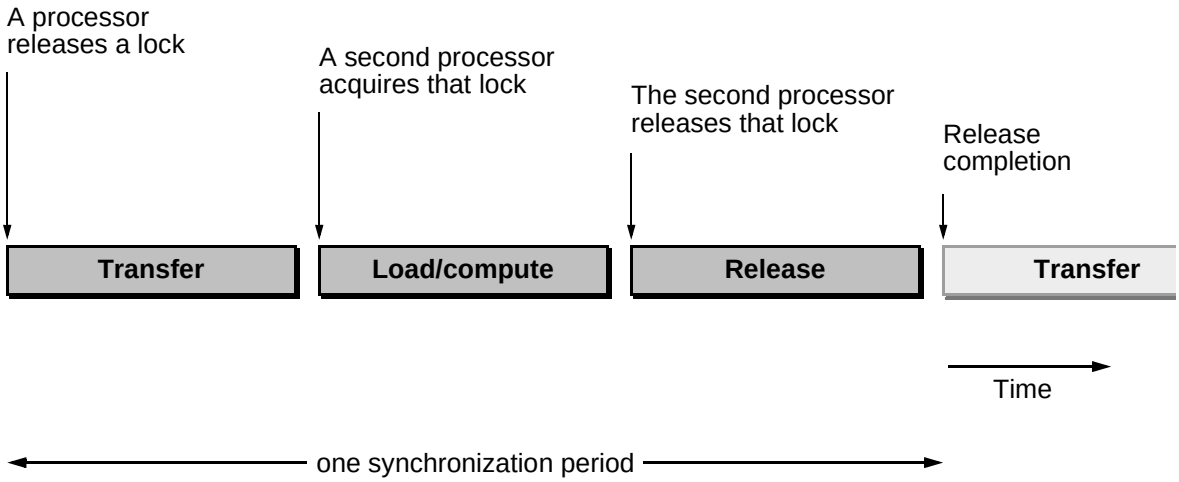



**Figure 1.2 Synchronization period.** The synchronization period consists of three phases: transfer, load/compute, and release. The depicted scenario assumes a contended lock.

put of synchronization operations achievable and, thus, may play an important role in determining scalability for programs that rely heavily on locks.

The synchronization period is not a device that can serve to predict a program's performance, but rather a framework in which to reason about synchronization inefficiencies and to evaluate qualitatively the benefits of optimizations. In particular, I will use the synchronization period to discuss the merits of the four locking mechanisms described in the next subsection.

### 1.2.2 Locking mechanisms

I identify a set of four previously proposed mechanisms that locks may incorporate to reduce the inefficiencies or time spent in the three phases of the synchronization period. They are *local spinning*, *queue-based locking*, *collocation*, and *synchronous prefetch*.

*Local spinning* permits a requesting node to spin on a local copy of the lock while waiting for it to be released, reducing traffic on the interconnection network. *Queue-based locking* builds a queue of processors waiting for a lock, reducing arbitration inefficiency and the time to transfer the lock. With this mechanism, arbitration is performed fairly and exactly once for each processor joining the queue. Primitives such as test&set arbitrate among all contending processors each time the lock is released: when the lock is freed, all the waiting processors attempt to acquire the lock collectively but only one will succeed. These failed attempts waste network bandwidth. Furthermore, schemes such as test&set do not guarantee fairness. Queue-based locking also provides the opportunity to optimize lock transfer by identifying ahead of time the lock releaser and the new acquirer. Therefore, the processor releasing the lock can forward it directly to its new destination. *Collocation* transfers critical data simultaneously with the lock, thus reducing the inefficiencies associated with accessing data protected by the lock. *Synchronous prefetch* allows a process to issue a request for a lock in advance of the critical section, hopefully overlapping useful computation with the lock transfer.

Using the synchronization period, I demonstrate how each mechanism reduces or even completely eliminates synchronization inefficiencies. This analysis induces a partial ordering of known synchronization primitives allowing a qualitative discussion of the relative merits of these primitives. This ordering is established by comparing the mechanisms that each primitive implements. Unfortunately, this framework is not universal since these mechanisms cannot

help reason about certain synchronization primitives such as fetch&add [GGK+83] and the load-reserve/store-conditional instructions [JHB87]. This framework successfully incorporates test&set, test&test&set, MCS locks, Anderson's locks [And89, And90], Graunke and Thakkar's locks [GT90], LH and M locks [MLH94], Lee and Ramachandran's scheme [LR90], QOLB, and message based locks [KCK99]. I classify each of these synchronization primitives according to which mechanisms they incorporate. An earlier presentation of this framework and classification scheme appears elsewhere [KBG97].

### 1.2.3 Performance of locking primitives

Using detailed simulation with both a microbenchmark and six parallel shared-memory applications, I measure the performance of six locking primitives: test&set, test&test&set, MCS locks, LH and M locks, and QOLB. Whenever possible and sensible, I extend these primitives with the mechanisms listed in the previous subsection, as well as with exponential backoff (a policy regarding actions taken when a process fails to acquire the lock). I also measure the performance of reactive synchronization schemes, which attempt to select dynamically the primitive best suited for a given level of contention for a lock. In the implementation that I study in this dissertation, the reactive synchronization chooses between test&set with exponential back-off for low contention phases and MCS for high contention phases.

I evaluate the performance of these different synchronization primitives by measuring the running time of both a simple microbenchmark and six shared-

memory applications. The microbenchmark demonstrate the performance potential of each locking primitive by measuring the elapsed time as a varying number of processors access a critical section a fixed number of times evenly distributed among the contenders. This microbenchmark is similar to the one used by both Anderson [And89, And90], and Lim and Agarwal [LA94]. In this test program, each processor repeatedly acquires a lock, waits a constant amount of time, releases the lock, and waits a bounded random amount of time before attempting to acquire the lock again. Under low contention, this microbenchmark illustrates the inefficiencies of acquiring and releasing a lock. With increasing levels of contention, this microbenchmark demonstrates the inefficiencies associated with transfering the lock from a processor to the next.

I also evaluate the performance of locking primitives by measuring the running time of six applications drawn from the SPLASH [SWG92] and SPLASH-2 [WOT+95] benchmark suites. These benchmarks are Barnes, Mp3d, Ocean, Pthor, Raytrace, and Water-Nsq. A subset of the performance measurements and analyses presented here appears elsewhere [KABG95, KBG97].

In addition to the evaluation of individual locking primitive, I also estimate the performance benefits that can be expected from each individual locking mechanism. I also quantify the benefit of exponential back-off.

A qualitative performance evaluation such as the one presented in this thesis may be highly dependent on assumptions that are valid at the time of this writing, but that may no longer be realistic in the future. To add to the robustness of these results, I perform some sensitivity analysis running some key experiments

using two different sets of assumptions. One set is representative of the current state of technology, while the second set is an attempt to model technology as it is likely to stand five years hence.

The performance of locking primitives is particularly sensitive to memory and interconnection network latencies. Indeed, synchronization primitive operations consist mostly of memory accesses to acquire and to release a lock and message transmissions to transfer the lock from a processor to another. The relative speeds of processors, memory units, and interconnection networks has changed dramatically over the years. For example, during the past three decades the speed at which a processor can initiate instruction execution has improved much faster than the speed at which a memory system can supply data (as illustrated in Figure 1.3). This trend has created a gap that reveals the growing importance of minimizing the number of memory accesses. Similarly, network latencies (dealing with even greater physical distances) are unlikely to match the pace set by the rate of instruction execution in modern processors. In the same way, this other trend stresses the importance of minimizing the number of messages sent on the network. Therefore, the synchronization inefficiencies are likely to grow worse, lending increasing importance to efficient locking primitives. Indeed, the results presented in this thesis corroborate this analysis.

### 1.2.4 Implementation of locking primitives

The synchronization period allows us to understand the inefficiencies associated with mutual exclusion and to understand the requirements to achieve syn-

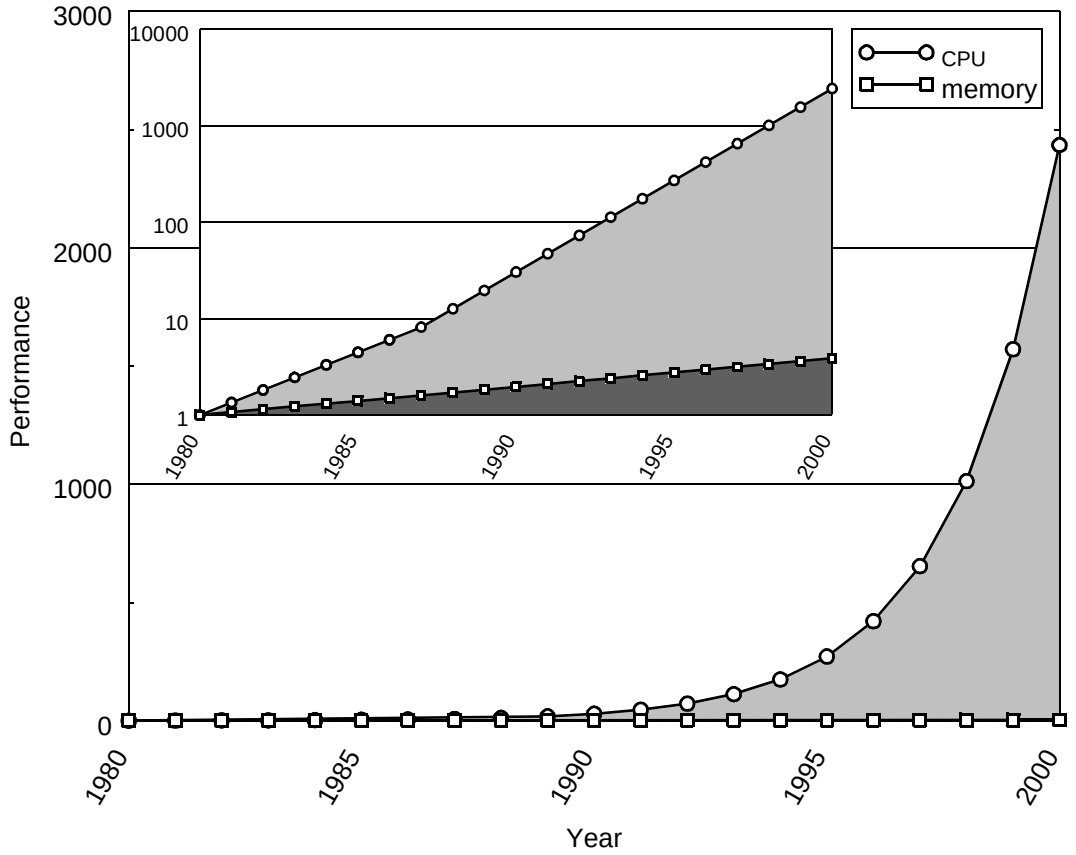



**Figure 1.3 Relative performance of CPU and memory plotted over time.** The original data for this figure appears in the second edition of Computer Architecture: A Quantitative Approach [HP95]. The memory baseline is 64-KB DRAM in 1980, with three years to the next generation and a 7% per year performance improvement in latency. The CPU line assumes a 1.35 improvement per year until 1986, and a 1.55 improvement thereafter. Note that the DRAM improvement is barely noticeable. The vertical axis must be on a logarithmic scale (as shown in inset) to record a change.

chronization with minimal impact on performance. However, the full support for such very efficient synchronization may be prohibitive. I address this issue by proposing and discussing a range of implementations that represent different discrete points in the performance/cost spectrum. In particular, I identify six hardware mechanisms required to support each of the four locking mechanisms. These hardware mechanisms are *naming*, *protocol processing*, *synchronous cache-to-*

*cache transfer*, *place holder allocation*, *non-blocking instructions*, and *association of lock and data*. For each of the six hardware mechanisms I discuss several implementation alternatives. Some of these alternatives can be implemented with today's processors; while other proposed implementations require modifications to the microprocessor or its external interface. While attempting to evaluate the actual cost of each proposed alternative is plainly impossible, I still attempt to evaluate the relative cost of each implementation.

Finally, I demonstrate that it is feasible to support on today's hardware an efficient synchronization primitive that includes all four locking mechanisms. To this end, I describe and evaluate SOFTQOLB, a software implementation of QOLB on a departmental cluster of workstations that consists of unmodified dual processor Sun SPARCstation 20s, each with two 66-MHz HyperSPARC processors [ROS93].

### 1.2.5 VAQUM

Implementations of the shared-memory paradigm achieve efficiency through data replication in caches associated with each processor in the system. A coherence protocol maintains consistency of the contents of these caches. As a rule the protocol maintains coherency at a granularity of a fixed cache line size. Locking primitives that support collocation operate on variables stored in these cache lines. Therefore, the size of the cache line constrains the collocation strategy and the performance gain expected from collocation. For instance, if the critical data does not fit in a single cache line, access to the part of the data that does not reside in the same cache line as the lock may lead to undesirable cache misses.

To overcome this limitation, I propose a new locking primitive, VAQUM, where the user specifies what the critical data are and when their access must be exclusive. VAQUM enforces mutual exclusion and efficiently ships *all* the critical data to the next processor, if any. I implement VAQUM as an extension to SOFTQOLB, the efficient synchronization implementation discussed earlier.

The technique of sending related data as a unit naturally extends to any data structure in a program, not just those accessed through a critical section. In essence, this technique allows the user to select the cache coherence granularity best suited for the access patterns to each data structure allocated in memory and provides efficient support for synchronization if the program requires exclusive access to any such data structure.

In summary, this extension combines the performance benefit of efficient synchronization with the flexibility of selecting the sharing granularity to match best the access patterns of different data structures allocated by a program.

I describe and evaluate CLEAN, a distributed shared-memory system that implements VAQUM and allows the user to specify, for specific regions of memory, the granularity at which cache consistency should be kept. For ease of implementation, CLEAN restricts the possible values of grain sizes to be any power of two between 32 and 2,048 bytes (inclusive). To avoid complications, CLEAN's memory allocator handles all allocation requests for shared-memory. In particular, the allocator rounds each request to the nearest power of two that is larger than the request. The difference between the request and the actual region of memory allocated is left unused (but still transferred with the used part of the object).

## 1.3 Historical perspective and related work

This section briefly reviews previous and parallel work related to the contributions presented in this document. More detailed reviews and comparisons to related work appear as necessary in each chapter.

### 1.3.1 Synchronization primitives

Test&set was the only synchronization primitive available on numerous early systems (such as the IBM 360 series [ABB64]). Originally intended to help coordinate processes running multiplexed on time-shared uniprocessors, test&set was later also used as a synchronization primitive on multiprocessor systems. Although it performs adequately in the absence of contention, test&set is very inefficient in a multiprocessor under heavy load. Indeed, when two or more processors are waiting to obtain a lock, test&set can generate an unbounded amount of traffic on the interconnection network. This observation led Rudolph and Segall to propose test&test&set [RS84], which reduces the load on the network by having waiting processors spin on a local copy of the lock. Local spinning bounds the traffic generated to at most $O(N)$ messages per locking transaction, where $N$ is the number of processors attempting to get the lock.

Test&test&set still has the potential to generate a lot of unnecessary traffic. Even worse the destination of all that traffic may be a single node (the “hot spot”) leading to congestion that can degrade the performance of the entire system [PN85]. To mitigate the impact of hot spots, Rudolph and Gottlieb first proposed combining requests in the interconnection network [Rud81]: when two requests

to the same memory location meet in a switch competing for the same output port, the switch sends a single combined request in place of the two conflicting requests; the switch later creates two separate replies based on the reply to the combined request. The two key benefits of this technique are that it has the potential to satisfy $N$ requests in $O(1)$ time and that its effectiveness increases with network load. Applying this technique to the synchronization primitives reviewed so far, it is easy to see that test&set is combinable. In addition to test&set, many other types of requests are combinable [KRS86]. Among them, the two most widely studied combinable requests are probably fetch&add (add a constant addend to a memory location and return the old value stored herein) and fetch&increment (a special case of fetch&add, where the only permitted addends are ±1) [GK81, GLR83, GGK+83, PBG+85, KRS86, RCCT90, FG91]. Both of these primitives can implement mutual exclusion (they can be seen as generalizations of test&set) but also form the basis of highly concurrent implementations of important coordination problems such as the readers-writers problem and the queue data structure [GLR83].

Fetch&add in conjunction with combining has the potential to reduce the impact of hot spots and permits highly concurrent data structure implementations, but this approach suffers also from a number of drawbacks. Obviously, fetch&add complicates the design of the network switches and the memory modules, which may slow down traffic. But, perhaps the most serious limitation of this proposal is that it omits the concept of caching. Although, cache-coherent shared-memory systems can support an instruction like fetch&add; I am not

aware of any proposal that integrates coherent caches and combinable synchronizing instructions successfully. The reason for the lack of a successful integration, so far, may stem from the conflicting nature of request combining and caching. On one hand, request combining is inherently a non-local phenomenon: it is most effective where requests converge, on their way to some destination, away from their sources. On the other hand, caching is a mechanism that attempts to eliminate remote communication entirely by keeping copies of frequently accessed data in a fast local memory. An integrated solution requires the reconciliation of these two conflicting requirements.

While combinable memory requests such as fetch&add attempt to reduce the impact of (synchronization) traffic in the interconnect, distributed queue-based locking primitives attempt to minimize the number of network transactions required to acquire and release a lock to $O(1)$ messages. These latter primitives maintain a queue of waiting processors, in which each node maintains pointers to adjacent processors in the queue. These primitives minimize network traffic, (1) by performing arbitration among contending processors as each processor enters the queue, (2) by allowing each waiting processor to spin locally on a copy of the lock (as with test&test&set), and (3) by restricting the number of nodes involved in the lock transfer.

Goodman, Vernon, and Woest first describe such a queue-based locking primitive, which they call QOLB[1] [GVW89]. QOLB (pronounced “Colby”) maintains the queue of waiting processors in hardware storing pointers to adjacent queue

1. Originally called QOSB.

entries in fields associated with each cache line (thus extending each cache line state). When a processor requests a lock, it first allocates a cache line (containing initially stale data) and then sends a request to join the queue. The processor waits for the lock by spinning locally until the cache line contains valid data sent from the previous lock owner. When the holder releases the lock, it sends the corresponding cache line directly into the next processor's cache, thus transferring the lock in exactly one network message.

Since QOLB defers the transfer of the cache line holding the lock until the processor explicitly releases it, it is constructive to place data accessed in the critical section protected by the lock in the same cache line as the corresponding lock. In effect, this optimization allows useful data to be transferred along with the lock, thus reducing or possibly eliminating cache misses completely during the execution of the critical section. Therefore, the collocation of lock and protected data in the same cache line reduces network traffic and shortens the time a processor holds a lock, thus decreasing lock contention. Finally, QOLB is a non-blocking primitive; therefore a program can request access to a lock ahead of time, hopefully further decreasing a program's running time by overlapping lock transfer time with other useful computation.

Inspired by QOLB, the Stanford DASH multiprocessor prototype implements a queue-based synchronization primitive [LLG+92]. However, instead of storing the queue in a distributed manner across the caches in the system, the prototype stores the queue at the directory. This organization leads to a simpler design, but also introduces a level of indirection that increases the lock transfer time. Thus,

the lock can no longer be transferred directly from the releaser to the subsequent acquirer; instead the lock must always go through the directory. Also this primitive names synchronization variables in an address space separate from the regular shared-memory address space. In the current implementation, there is no possibility to establish a correspondence between a synchronization variable and associated data precluding collocation.

VAQUM, proposed in this thesis, is an extension of QOLB that improves the handling of protected data associated with a lock. In a single network message, QOLB can transfer a lock and its associated data by allocating those data and the lock in the same cache line purposefully. The (usually fixed) size of the cache line limits the effectiveness of this technique. The size of protected data may be much smaller than a cache line size leading to messages carrying mostly useless information. Alternatively, exclusive data may overflow a single cache line, in which case additional cache misses caused by references to overflowed data will slow down the critical section execution. VAQUM optimizes the transfer of the lock and associated data by selecting the message size most appropriate for a given critical section. If the amount of data associated with a lock is large, VAQUM will select a message large enough to accommodate all of the protected data. Thus, VAQUM will ensure data delivery at that the destination in one logical unit; as a result the processor will not incur misses while executing in the critical section. On the other hand, if the program associates little data with the lock, VAQUM will select a small message, thus speeding up the lock transfer. In the extreme, a program may not collocate any data with the lock at all, in which case VAQUM degenerates

into a primitive resembling message-based locks that I review later in this section.

Lee and Ramachandran propose an extension of QOLB that supports read locks [LR90]. However, their technique only applies to bus-based systems without obvious ways to extend it to other systems with arbitrary network topologies.

Anderson [And89, And90], and Graunke and Thakkar [GT90], independently describe queue-based locking algorithms implemented entirely in software. Subsequently, Mellor-Crummey and Scott describe improvements to Anderson's algorithm in related papers [MCS91a, MCS91b]. These proposals allocate data structures in shared memory and insert processors in lists or circular arrays using atomic instructions such as swap or compare&swap to update the concurrent data structures correctly. The price of maintaining the queue in software is somewhat larger inefficiencies. A synchronization period still requires only $O(1)$ messages to complete, but instead of just one message (as is the case for QOLB or VAQUM), the critical path of a lock transfer requires typically two global shared-memory operations [MLH94]. Also, since each waiting processor spins on a distinct memory address, exclusive data cannot be effectively collocated with the lock. Finally, these approaches cannot easily prefetch accesses to a lock effectively. The difficulty resides in the fact that these algorithms require different numbers of memory accesses to acquire a lock depending on the state of that lock. If a lock is busy, at least two memory accesses are required to obtain the lock, the first of which needs to complete before the second access can be issued definitely.

While a process could easily prefetch the first access, coordinating the prefetch of the second memory access is more difficult to accomplish.

Numerous extensions to these basic software queueing algorithms have been proposed. Magnuson, Landin, and Hagersten [MLH94] and Craig [Cra93] discuss algorithms that have a slightly better cache behavior under medium contention. Also, none of the synchronization primitives studied in this thesis are robust in the presence of failure; in other words a process that fails to release a lock for some reason (e.g., the user is running an incorrect program or one node of the multiprocessor system has failed) may prevent the forward progress of other processes. Strategies have been developed that aid in isolating the problem or in recovering from it. Bohannon, Lieuwen, and Silberschatz present a recovery strategy [BLS+95]. They extend the queue-based algorithm proposed by Mellor-Crummey and Scott such that it allows a system to recover a lock held by a process believed to have failed.

All the queue-based synchronization primitives considered so far assume either special hardware support for queue-based locking (e.g., QOLB) or implement a queue in software using a combination of memory accesses (which the underlying system satisfies transparently by communicating with other nodes using messages) and hardware synchronization primitives such as test&set. Another solution is to design synchronization primitives based on top of a message passing interface directly. This solution does not require any special hardware beyond the ability to send and receive messages. Message-based synchronization uses lock managers to control lock ownership. A processor acquires a lock by sending a

message to a lock manager. Then, the lock manager either queues the request if the lock is busy or satisfies the request immediately. To ensure atomicity, a lock manager processes each request to completion one at a time. Message-based locking supports local spinning by having synchronization requestors wait for a granting message from the lock manager without generating additional network traffic. Message-based locking could also implement synchronous prefetch by allowing a program to request a lock ahead of time. Message-based synchronization does not support collocation unless the underlying messaging system is integrated with the coherence protocol. FLASH [KOH+94] and Tempest [RLW94] are two systems that support such an integration. The Stanford FLASH Multiprocessor is a high-performance, scalable parallel computer designed around a programmable memory controller to support both message passing and shared memory efficiently. Tempest is a collection of mechanisms allowing programmers and compiler writers to develop portable parallel code that can exploit the advantages of shared memory, message passing, and hybrid combinations of the two.

Two organizations for a message-based lock manager are possible: a centralized solution or a distributed solution. In the centralized solution, all lock requests are routed to a special node (the manager[1]), which supervises accesses to the locks. If a node wants to enter a critical section, it sends a request to the lock manager. The lock manager then grants the lock to the requestor by sending it a message. When the node leaves the critical section, it sends another message to the lock

1. To avoid congestion, the system has multiple managers distributed among the nodes in the system.

manager; as soon as the manager receives that message, it can service the next processor in line, if any. In most cases, this solution requires two messages on the critical path to transfer the lock. In contrast, the implementation of a distributed lock manager attempts to reduce this number of messages to one under high contention. It achieves this goal using an implementation that is similar to the organization of QOLB: each node stores pointers to adjacent nodes in the queue. However, this improved high-contention behavior comes at a price: access to an idle lock will require typically three messages.

Requiring fewer messages on the critical path to transfer a lock than other software queue-based locking algorithms, message-based synchronization primitives should perform better. However, this solution may not work in systems where a message passing abstraction is not available. Such systems may include some bus-based computers in which inter-processor communication has traditionally taken place through shared-memory.

To my knowledge the first implementation of a message-based locking primitive for a distributed shared-memory system appears in TreadMarks [KCDZ94]. TreadMarks is a distributed shared memory system developed at Rice University [ACD+96]. In two related papers [KCK98, KCK99], Kuo, Carter, and Kuramkote discuss several message-based lock implementations. They evaluate both user-level and in-kernel lock managers as well as three lock manager organizations. They find the inefficiencies of context switches to be significant and therefore suggest using in-kernel managers. The three lock manager organizations the authors discuss are a centralized solution, a distributed solution, and an adaptive solu-

tion that reverts to the centralized solution when there is little contention. Blizzard, an implementation of the Tempest interface that runs on a cluster of Sun SPARCstation 20s, supports a centralized message-based lock [SFL+94]. In his dissertation [Hei98], Heinlein describes a distributed message-based lock implementation for the Stanford FLASH multiprocessor.

Lim and Agarwal observe that different locking algorithms are better suited for different levels of contention. When there is no contention, test&set can quickly acquire and release a lock, but suffers intolerable latencies when multiple processors try to acquire a lock concurrently. Alternatively, queue-based software primitives provide a robust solution under contention, but suffers unnecessary delays in the absence of congestion (even if the cache line that contains the lock is available locally, the lock acquisition will still require executing ten instructions or more [MLH94]). Therefore, Lim and Agarwal proposed reactive synchronization, a policy that tries to select the software primitive best suited for a perceived level of lock contention [LA94].

In the context of their work on cooperative shared memory, Wood and his colleagues [WCF+93] describe cooperative prefetch, a mechanism that is similar to synchronous prefetch. Cooperative prefetch permits a processor to request data ahead of time and lets the current owner delay honoring the request until the current owner no longer needs the data. A processor triggers the data transfer with a call to a special primitive dubbed `check_in`. Note, however, that cooperative prefetch is not a synchronization primitive, but rather a hint to help the hardware decide when and where to transfer data.

Another approach to lock-based synchronization taken by Jensen, Hagensen, and Broughton [JHB87] is to define a set of primitive instructions from which to build synchronization primitives. Jensen, Hagensen, and Broughton proposed load-reserve and store-conditional, two instructions that can implement common read-modify-write operations, such as test&set, swap, and compare&swap. The ideas behind these instructions is to expose the steps involved in performing read-modify-write operations (i.e., first read, then modify, and finally write). By being able to specify the modify step, a program can compose its own atomic primitive. A program uses this pair of instruction as follows. First, it issues a load-reserve that loads a value into the processor and sets a reservation bit. Second, the program modifies this value as it sees fit. And finally, the program conditionally writes the modified value back to memory. If the reservation bit is still set the store succeeds, otherwise the processor cancels the write operation. Upon detecting failure, a program will typically retry the sequence of instructions after waiting for a while. Events that clear the reservation bit include accesses to the same memory location by other processors in the system. The cache coherence protocol can easily detect such concurrent accesses to the same memory location.

Since their introduction, several instruction sets have adopted these instructions: the MIPS instruction set architecture starting with the second revision [KH92] (`load linked`, `LL` and `store conditional`, `SC`), the Alpha processor architecture [Sit92] (`load quadword locked`, `ldq_l` and `store quadword conditional`, `stq_c`), and the PowerPC instruction set [MSSW94] (`load word and reserve indexed`, `lwarx` and `store word conditional indexed`, `stwcx.`[1]).

Herlihy and Moss [HM93], and Stone and his colleagues [SSHT93] independently proposed a generalization of the load-reserve and store-conditional instructions that handles multiple words instead of just one. They view a critical section as a transaction that a processor can execute optimistically: read values, compute, write values and finally commit the new values if no conflict is detected.

A very attractive property of the load-reserve/store-conditional instructions (and transactional memory) is that they allow programmers to construct *wait-free* concurrent data structures [Her91]. A concurrent object is wait-free if any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes (including failure). Wait-free programs avoid common problems with conventional techniques such as priority inversion and convoying.

However, the wait-free property does not hold unless the implementation of the load-reserve/store-conditional can guarantee that a reservation bit is cleared *if and only if* the corresponding memory location is written. Unfortunately, in practice, events other than shared memory write can nullify a reservation. For instance, the reservation bits are typically not part of a process state forcing the operating system to clear them on context switches.

Furthermore, Attiya, Lynch, and Shavit [ALS94] have shown that there exists a $\Omega(\log N)$ time gap between the performance of wait-free and conventional algorithms for at least one important problem. They prove time complexity lower

1. The extra period at the end of the instruction mnemonic is not a typographical error. The dot indicates that the instruction modifies the condition register.

bounds for algorithms of a variant of the *consensus* problem assuming failure-free execution. In this problem, $N$ processors must all agree on a single value chosen from an input set. It is trivial to construct a conventional algorithm that solves this problem in overall time complexity $O(1)$. They prove a lower bound of $\log N$ on the time complexity of any wait-free algorithm solving this variant of the consensus problem. Using the I/O automata formalism [LT89], they demonstrate that the synchronized schedules of each process must execute at least $\log N$ steps.

Several of the techniques presented so far (e.g., test&set and load-reserve/store-conditional) may fail to acquire a lock and thus must retry the unsuccessful operation. To avoid congestion and lack of forward progress, and to maximize the success rate of lock acquisition, it may be beneficial for a processor not to attempt again acquiring a lock immediately. Agarwal and Cherian first apply back-off techniques to synchronization primitives [AC89]. They use adaptive back-off methods to reduce the impact of invalidation traffic generated by software implementations of barrier synchronization. ALOHA, a radio-based, packet-switched network, first introduce back-off techniques [Abr70]. ALOHA proposes to perform fair arbitration and bandwidth allocation stochastically. Specifically, if nodes detect other transmissions while sending a packet, each of them backs off for a random interval before attempting a retransmission. Ethernet refines the back-off techniques by having the collision history influence the length of the random interval: the more collisions that occurred recently the longer the wait [MB76]. Anderson first applies the exponential back-off techniques to locking primitives [And90]. He proposes two variants of test&test&set with exponential backoff

which differ in the way the adaptive delay is applied. In one case the primitive inserts a delay after each failure to acquire the lock; in the other case the primitive waits between each reference to the lock.

A major concern of synchronization is forward progress or lack thereof. For instance, it is trivial to construct an example with the load-reserve/store-conditional instructions that fails to make forward progress. For example, some processors implementing these instructions clear the reservation bit on a TLB refill. Consider the case of a direct-mapped TLB and assume that a lock and the variable it protects live in distinct pages yet map into the same TLB entry. The access to the lock at the beginning of the critical section fills the TLB and sets the reservation. The subsequent access to the variable in the critical section suffers a TLB miss forcing a refill and therefore the cancellation of the reservation. The most amazing part of this example is that it requires only a single processor. Note that even exponential back-off will not help in this case. Another example of lack of forward progress involves three processors. In this example, two processors accessing a lock continuously using test&set may prevent a third processor from ever getting the lock. But in this case, exponential back-off alleviates this problem by spreading each processor's request.

The previous paragraph mentions a serious drawback of the load-reserve/store-conditional instructions. Yet, in spite of this problem, many modern processors implement these instructions. These instructions offer versatility and design advantages that are considered important enough to justify their inclusion in an implementation. First, these instructions are extremely versatile, being able to

emulate many popular atomic operations such as test&set, swap, or compare&swap. Second, these instructions are far easier to implement in hardware than conventional atomic memory operations. To the microprocessor pipeline, these instructions look like any other load or store instruction with the simple addition of a reservation signal informing the memory system of the special nature of the access. On the other hand, atomic memory operations require a state machine to sequence through the access to the original value, the computation of the new value, and the storing of the new value. The addition of the state machine complicates the pipeline design and may introduce new deadlock situations.

### 1.3.2 Framework/formalization

In related papers [KRS86, KRS88], Kruskal, Rudolph, and Snir propose a new formalism for read-modify-write operations. In this formalism, all read-modify-write operations take the form RMW($X$, $f$), where $X$ is a shared variable and $f$ is a mapping. RMW($X$, $f$) atomically computes $f(X)$, stores the result of the computation at $X$ and returns the value previously stored at $X$. They then use this formalism to extend fetch&add to arbitrary RMW operations (i.e., arbitrary $f$).

Herlihy defines a hierarchy of synchronization primitives such that no primitive at one level has a wait-free implementation based on primitives at lower levels [Her91]. This hierarchy finds atomic load and store instruction at level 0; test&set, swap, fetch&add at level 1; and memory-to-memory swap and compare&swap at level ∞. Thus, in the sense defined by this hierarchy, com-

pare&swap is a universal primitive that can simulate all the other primitives using a wait-free algorithm. Based on this hierarchy Herlihy concludes that fetch&add is not a universal primitive as conjectured by Gottlieb, Lubchevsky, and Rudolph [GLR83]. Herlihy constructs this hierarchy by reducing the question "is there a wait-free implementation of synchronization primitive X by primitive Y" to the question "what is the maximum number *N* of processors for which the synchronization primitive can solve the simple consensus problem."

These previous formalisms pertain to extending or reasoning about the capabilities of synchronization primitives. In this thesis I decompose synchronization primitives into mechanisms that they implement. This decomposition defines a framework in which to reason about performance and implementation of synchronization primitives:

- This framework allows for a qualitative performance comparison of two synchronization primitives by contrasting the mechanisms they implement.
- This framework allows the separate estimation of the performance merits of each mechanisms.
- Finally, keeping in mind the performance merit of each mechanism, this framework allows the evaluation of implementation cost of each mechanism.

### 1.3.3 Evaluation

Numerous performance evaluations of synchronization primitives have appeared in the literature. I restrict this discussion to the most relevant studies. In almost all cases the experimental setups are different enough that they pre-

clude the direct quantitative comparisons between results published in the literature and results found in this report. Fortunately, in most cases a qualitative comparison is possible and given.

Using a microbenchmark, Anderson [And89, And90] compares the performance of test&set, test&test&set, test&test&set with exponential back-off and his queue-based locking algorithm on a Sequent Symmetry Model B [LT88]. Sequent Symmetry is a bus-based shared-memory multiprocessor; Model B uses 16MHz Intel 80386 processors. Anderson finds that test&set and test&test&set perform well under low contention (up to five processors) but their performance quickly degenerates for higher levels of contention. He also finds that exponential back-off helps provide reasonable performance across all levels of contention. Finally, under high contention, his queue-based locking algorithm outperforms all the other primitives he considers; however, the additional instructions required to maintain the queue cause its performance under low contention to be worse than these other primitives. These results are qualitatively consistent with the results I report in this thesis. Anderson also studies several variants of test&test&set with exponential back-off. His study discusses a total of four variants of test&test&set that represent the combinations spanning two dimensions: whether the delay is inserted between each reference to the lock or after failure to acquire the lock, and whether the delay is set statically or is adaptive (exponential). Among these four alternatives, his results indicate that exponential back-off with a delay inserted between lock references (the variant used in this thesis) is the best: it outperforms the other three solutions under low contention and its

performance under high contention is only slightly worse than the best of the other back-off alternatives. I do not measure Anderson's queueing scheme, but measure MCS instead. Both algorithms are very similar and I believe that they would display identical trends across varying levels of contention. Furthermore, Aboulenein and his colleagues [AGGW94] show analytically that Anderson's lock performs no better than MCS, as do Magnusson and his colleagues [MLH94]. Mellor-Crummey and Scott's measurements [MCS91b] confirm this analysis in one case, but disprove it in another case. This discrepancy may stem from aspects of a real system that are not captured by the analytical model such as the ability to exploit concurrency among operations in these algorithms. However, Mellor-Crummey and Scott's results also indicate that their algorithm scales better with increasing contention and that, empirically, Anderson's algorithm is at most 15% faster than their solution.

Using a microbenchmark, Graunke and Thakkar [GT90] compare the performance of test&set, test&test&set with exponential back-off, and their queue based lock on a Sequent Symmetry Model C bus-based multiprocessor, which uses 20MHz Intel 80386 processors. Graunke and Thakkar's microbenchmark is somewhat different than the one Anderson uses: they replace the random delay after each lock release with a constant delay and they increment a variable inside the critical section instead of holding the lock for a constant amount of time. In spite of these small differences, their results are similar to Anderson's and my results are consistent with them both. Graunke and Thakkar also study an optimistic version of test&test&set, a variant of test&test&set with exponential back-

off, and two implementations of a tournament lock. Optimistic test&test&set assumes that there is no contention: it attempts a test&set first and, if it fails, reverts to the traditional test&test&set (check the lock first and, if free, attempt test&set). Optimistic test&test&set performs as well as test&set under low contention and performs as well as test&test&set under high contention. As Anderson does, Graunke and Thakkar study two implementations of test&test&set with exponential back-off and draw the same conclusion: they find that a delay inserted between each reference to the lock performs better than a delay introduced after failure to obtain the lock. Finally, they also consider two tournament lock implementations. While both implementations scale well with increasing contention, they do not perform as well as test&test&set with exponential back-off (both variants) and the queue-based algorithm.

Measuring the running time of a simple microbenchmark on a BBN Butterfly 1 and on a Sequent Symmetry Model B, Mellor-Crummey and Scott compare the performance of their queue-based lock with, among others, test&set, test&test&set, test&set with exponential back-off and Anderson's algorithm [MCS91a, MCS91b]. The BBN Butterfly 1 is a shared-memory multiprocessor using a multistage interconnection network to connect processors and memory modules [HP95]. Mellor-Crummey and Scott's microbenchmark is different than Anderson's or Graunke and Thakkar's: it does not insert any delay at all; each processor repeatedly acquires and releases a lock without waiting.[1] However,

1. In one experiment, they introduce a constant delay in the critical section to be fair with their implementation of Anderson's algorithm on the Symmetry; they compensate for this delay in the presentation of their results [MCS91a].

despite these dissimilarities, Mellor-Crummey and Scott's experiments confirm the results already obtained by Anderson, and Graunke and Thakkar. Mellor-Crummey and Scott also show that MCS outperforms Anderson's algorithm on the Butterfly and that, under high contention, Anderson's algorithm is about 15% faster than MCS on the Symmetry. Mellor-Crummey and Scott also discuss test&set with linear back-off and several variants of ticket locks (analogous to the algorithm used in a bakery to ensure fair service with numbered tickets representing each customer's position in the queue). Their results show that test&set with linear back-off is not substantially better than test&set. They also show that ticket lock is not competitive with other queue-based algorithms unless it is associated with a form of adaptive delay (e.g., a delay proportional with the difference between the customer number and the number currently served). However, even with the adaptive delay, ticket lock does not perform nearly as well as MCS.

Using NWO, an accurate multiprocessor simulator, Lim and Agarwal [LA94, Lim95] compare the performance of their reactive synchronization scheme against test&set and test&test&set (both with exponential back-off), and MCS. NWO is an accurate cycle-by-cycle simulator of the Alewife multiprocessor [ABC+95], a shared-memory multiprocessor built at MIT. They analyze the performance of these different synchronization primitives measuring the performance of the same microbenchmark that Anderson uses, and two shared-memory applications: Mp3d and Cholesky. With the microbenchmark, they confirm the results already obtained by Anderson, Graunke and Thakkar, and Mellor-Crummey and Scott. Lim and Agarwal also show that their implementation of reactive synchro-

nization tracks well the performance of test&set with exponential back-off under low contention, and the performance of MCS under high contention. Comparing their shared-memory application results with mine is more difficult as our sole common benchmark is Mp3d, and even in that case, we use different input sizes. Overall, they find that reactive synchronization tracks the performance of the best performing primitive well. These results are consistent with the results reported here.

On the BBN GP1000 and TC2000, Zhang, Castañeda, and Chan [ZCC94] compare the performance of test&set, test&set with exponential back-off, ticket lock with proportional back-off, and MCS using a microbenchmark similar to the one used by Mellor-Crummey and Scott. Both the BBN GP1000 and TC2000 are two large-scale shared-memory multiprocessor systems with multistage interconnection networks connecting processors and memory modules; both systems are successors to the BBN Butterfly 1. Their results agree for the most part with those published by Mellor-Crummey and Scott, with one exception: the performance of test&set with exponential back-off on the GP1000 is, for the most part, worse than test&set. They attribute this behavior to higher network contention in the GP1000 than in the Butterfly 1.

In his thesis [Hei98], Heinlein compares the performance of his distributed message-based locking primitive with the performance of MCS and a lock implemented with the load-reserve/store-conditional instructions. To compare these primitives, he simulates the execution of a microbenchmark that is similar to the one Anderson uses and two shared-memory applications (Barnes and Water-

Nsq[1]) on a detailed simulator of the Stanford FLASH multiprocessor. His implementation of the load-reserve/store-conditional lock uses exponential back-off to reduce the impact of contention. His microbenchmark results show that all three considered primitives perform equally well under low contention (from two to four nodes). Past four nodes, however, the load-reserve/store-conditional lock quickly deteriorates; while MCS and his locking primitive continue to perform equally well: for these two primitives, the microbenchmark running time increases only slightly with increasing numbers of contenders. In a separate experiment, using a microbenchmark similar to the one Mellor-Crummey and Scott use (i.e., Anderson's test program without delays), Heinlein shows that, under high contention, his lock implementation transfers the lock much more quickly (60% faster) than MCS does. The reason this speedup is not noticeable in his experiment using Anderson's test program is that the microbenchmark spends a large fraction of its time idle inside the critical section. Heinlein's results with the two shared-memory applications indicate that his lock primitive performs best and load-reserve/store-conditional with exponential back-off performs worst. His results also indicate that both applications using any of the studied locking primitives scale reasonably well up until 64 nodes. Past that point, only his message-based lock is able to improve the running time of both applications independently of the input sizes he uses.

1. Heinlein also uses Ocean for his experiments; but with this application he focuses on the analysis of the performance of barrier synchronization.

Using an execution driven simulation, Kuo, Carter, and Kuramkote [KCK98, KCK99] compare the performance of test&test&set, MCS, and three implementations of a message-based lock. The three implementations of the message-based lock are a centralized solution, a distributed solution, and an adaptive solution that reverts to the centralized scheme when there is little contention. In their experiments, they simulate the execution of five programs: Mp3d, Barnes, Radiosity, Raytrace, and Spark98. The first four benchmarks are drawn from the SPLASH-2 [WOT+95] and Spark98 is a sparse matrix kernel [OSG98] that computes a sequence of matrix vector products. They run their simulations for multiple system arrangements: they vary the number of processors from 4 to 32 and they simulate two network configurations. I will focus this discussion on their largest system with the faster network since I simulate the performance of benchmarks only on a 32-node system and their fast network seems to model my choice of parameters more accurately. They find that MCS always outperforms test&test&set; MCS performs more than 30% faster than test&test&set for Barnes, Mp3d, and Raytrace which agrees with my results. They also find that message-based locks outperforms both MCS and test&test&set, and that the distributed lock performs better than the centralized solution except for Mp3d. This benchmark has very low lock contention, which favors the centralized solution. Finally, they find that adaptive or reactive solution tracks the performance of the best message-based solution well.

Goodman and Woest [WG91] present a comparative evaluation of test&set, MCS, and QOLB using both qualitative and quantitative techniques. Both techniques

rely on an ideal model of a system resembling the Wisconsin Multicube [GW88] assuming infinite caches and queues, and infinitely fast processors. Using a simple counting technique, they compare the idealized latency required to execute a critical section. They perform this comparison with four different sets of assumptions representing the combinations spanning two dimensions: (1) whether the critical section is empty or increments a counter and (2) whether the critical section is idle (and the lock is not local nor at memory) or busy. Their results show that QOLB outperforms all the alternatives in all cases. Also, they show that test&set outperforms MCS when the critical section is idle, and the opposite when the critical section is busy. These results are consistent with my results. In addition, they discuss a variant of test&set that operates at memory (i.e., uncached). They show that the at-memory test&set primitive performs only slightly worse than the coherent test&set when the critical section is idle and performs better than MCS when the critical section is busy. The reason this primitive performs so well under contention is that it prevents the lock from bouncing among caches when it is released. Exponential back-off achieves a similar effect stochastically.

In related papers [AGGW92, AGGW94], Aboulenein and his colleagues compare the performance of Anderson's algorithm, MCS, and QOLB analytically for the specific case of the SCI cache coherence protocol [IEE93]. They count the number of messages and memory operations required on the critical path for two situations that Goodman and Woest also used in their analysis. The first situation considers the execution of an idle critical section (assuming the corresponding lock is not local nor at memory); the second one examines the transfer of a contested lock.

For these two scenarios and considering either message count or memory accesses, they find that QOLB outperforms both MCS and Anderson's lock; and that MCS outperforms Anderson's lock. These results are consistent with the results I report in this document. In particular, we agree on the number of messages that it takes for either QOLB or MCS to transfer a lock from a processor to the next. However, we have a discrepancy in the number of messages necessary to acquire and release an idle lock. Compared to my results, their message count for MCS is pessimistic (2 more messages) and their message count for QOLB is optimistic (2 fewer messages).

Herlihy and Moss [HM93] evaluate the transactional memory model using PROTEUS [BDCW91], a high-performance, parallel-architecture execution-driven simulator developed at MIT by Brewer and his colleagues. Using three simple benchmarks, Herlihy and Moss compare their proposal against test&test&set with exponential back-off, MCS, QOLB, and the load-reserve/store-conditional instructions. They run the same experiments assuming two different types of machines. On one hand, they assume a bus-based system using Goodman's snoopy protocol [Goo83]; on the other hand, they assume an MIT Alewife-like machine using Chaiken directory protocol [ABC$^+$95]. The three benchmarks were (1) a counting benchmark, (2) a producer/consumer benchmark, and (3) a doubly-linked list benchmark. The first benchmark increments a variable a fixed number of times spread evenly among the nodes executing the program. This benchmark is somewhat similar to the microbenchmark used by Graunke and Thakkar except that it does not wait between accesses to the critical section. In the second

program, *N* processors share a bounded buffer, initially empty. Half of the processors insert items in the buffer, the other half retrieve items from the buffer. The program finishes when $2^{16}$ operations have completed. In the last benchmark, *N* processors share a doubly-linked list anchored at the head and at the tail. Each processor dequeues an item from the tail and then enqueues it by threading it onto the list at the head. The program terminates when $2^{16}$ operations have completed. They run the experiments with the load-reserve/store-conditional instructions differently for each benchmark. In the single word counting benchmark, the load-reserve/store-conditional instructions are used directly to increment the variable correctly. In the other benchmarks, which associate more than one memory word per lock, the load-reserve/store-conditional instructions are used to construct a spin lock. With one exception, their results indicate that transactional memory has substantially higher throughput than any of the other primitives. Indeed, transactional memory does not have explicit locks and therefore requires fewer memory accesses to complete the execution of a critical section. The exception to the dominating performance of transactional memory is the load-reserve/store-conditional instructions when they are used to update the counter directly. Direct use of these instructions requires no separate commit operation, saving an extra memory access as compared to transactional memory. In contrast, when these instructions are used to implement a spin lock, their performance drops considerably, on a par with the performance of MCS and test&test&set with exponential back-off. The performance of QOLB is generally second best, requiring typically one more memory reference than transactional memory but far fewer than

the other alternatives. They also find that test&test&set with exponential back-off performs in general worse than MCS on the simulated bus-based system, which is consistent with the results published in literature. On the other hand, their results indicate that in most cases test&test&set with exponential back-off performs better than MCS on the directory-based machine, which is not consistent with either Mellor-Crummey and Scott's or Lim and Agarwal's results. This difference may stem from their model of the network connecting the nodes in the system.

In this thesis, I evaluate the performance of test&set, test&test&set, MCS, LH, M, QOLB and reactive synchronization schemes on a simulated directory-based cache-coherent shared-memory multiprocessor. Where possible and sensible I extend these primitives with the collocation and prefetch mechanisms and the exponential back-off policy. In all, I evaluate a total of sixteen synchronization constructs. My results are in general qualitatively consistent with the results available in the literature with the two notable exceptions. Zhang, Castañeda, and Chan's results indicate that test&set with exponential back-off performs for the most part worse than test&set on the BBN GP1000. Similarly, Herlihy and Moss find that, under high contention, test&test&set with exponential back-off can perform better than MCS on their simulated directory-based multiprocessor system. These two behaviors are uncharacteristic of the results published in the literature and are inconsistent with my results. Zhang, Castañeda, and Chan claim that high network contention in the GP1000 interconnection network causes test&set with exponential back-off to perform poorly. Herlihy and Moss do

not give any reason to explain the poor performance of MCS in their simulated platform. However, our modeled systems are sufficiently different that I cannot compare our results qualitatively.

I also evaluate the performance of SOFTQOLB, my software implementation of QOLB, comparing it against test&set, test&test&set, MCS, and a centralized message-based lock. I perform this evaluation on a cluster of unmodified SPARCstation 20s. In general, this evaluation confirms results obtained on my simulated system and results published in the literature. I note, however, that the inefficiencies introduced in my software implementation of QOLB cannot realize the performance potential promised by an ideal implementation QOLB.

### 1.3.4 Implementation

Many shared-memory multiprocessor systems include specific special support for synchronization: compare&swap on the IBM System/370 architecture [CP78]; full/empty bits on the HEP multiprocessor [Smi81], on the Tera computer system [ACC+90], and the Alewife [ABC+95] multiprocessor; fetch&add on the NYU Ultracomputer [GGK+83] and the IBM RP3 [PBG+85]; QOLB on the Wisconsin Multicube [GW88] and the IEEE SCI standard [IEE93]; the primitives for locking and unlocking cache lines on the KSR1 [KSR91]; and a centralized queue-based lock and fetch&increment on the Stanford DASH prototype [LLG+92].

Scott describes the support for synchronization provided on the Cray T3E [Sco96]. This machine supports a shared memory address over distributed memory, but does not automatically replicate shared data in local caches. T3E supports

fetch&increment, fetch&add, compare&swap, and masked-swap. For each bit in the operand, masked-swap performs a swap only if the corresponding bit in the mask is set.

Cedar, a machine prototype built at the University of Illinois in Urbana-Champaign [KDL+93], supports special purpose synchronization primitives that are targeted towards the support of executables produced by an automatically parallelizing compiler. The project focuses on supporting the execution of large scientific programs well. Cedar exploits the parallelism found in loops of these large scientific applications. Processors execute different loop iterations and synchronization honors the proper data dependence as defined in the original sequential code. Zhu and Yew discuss this type synchronization support in details in their article [ZY87].

The first proposal for a cache coherence protocol with explicit support for synchronization is due to Bitar and Despain [BD86]. Their proposed bus-based protocol includes support for efficient spinning, data collocation, and prefetch; but not queueing. Both the Wisconsin Multicube [GW88] and the IEEE SCI also introduce special states in their coherence protocols to support QOLB. All these proposals assume that different coherence policies are required to support normal memory accesses and synchronization operations well. While additional states may improve performance, it also adds to the complexity of the coherence protocol.

Michael and Scott study issues in support important synchronization primitives that are popular on small-scale shared-memory systems but missing on most larger systems [MS95]. In particular, their work asks three questions. First,

where should atomic operations be executed: in the cache controllers, at memory, or both? Second, which cache coherence policy should be used for atomic accesses: no caching, write-invalidate, or write-update? Finally, what auxiliary functions, if any, can be used to enhance performance? The results of their study suggests that systems implement compare&swap in the cache controllers using a write-invalidate coherence policy. In addition, they recommend supporting a load-exclusive instruction. Load-exclusive reads data and acquires exclusive access. This instruction can be used instead of a conventional load to read data that is then accessed by compare&swap. This instruction also improves the efficiency of the coherence protocol when accessing migratory data [GW92].

In contrast to Michael and Scott study, this thesis does not ask which locking primitive(s) a system should support, but rather which locking mechanism(s). This thesis, then, goes on to describe what the alternatives are to support each of the four locking mechanisms in current and future multiprocessor systems. In particular, I identify six hardware mechanisms that are required to support the four locking mechanisms.

### 1.3.5 CLEAN

Work related to CLEAN, the distributed shared memory system that implements the efficient synchronization primitive called VAQUM, falls into three categories: (1) systems that support dynamic variable cache line sizes in hardware [DL92, JMH97, KW98]; (2) software distributed shared-memory systems [BZS93, Nik94,

JKW95, SL94a, SL94b, SGZ93]; and (3) software annotations to improve performance [LCW94].

Dubnicki and LeBlanc propose a coherent cache architecture that attempts to select dynamically a line size that suits best a particular region of memory, at a particular instant of a program's execution [DL92]. Their technique uses history information to decide whether to split a cache line (to avoid false sharing), or to merge two cache lines (to benefit from prefetching through spatial locality), or to maintain status quo. On average, their technique improves the running time of an application by 9% over the same application using the static line size that leads to the best running time.[1]

Johnson, Merten, and Hwu present a similar scheme for uniprocessors [JMH97]. Their goal is to optimize the use of the cache and the bus bandwidth. However, their results show that their scheme outperforms applications using the best static line size by at most 5%

In contrast to the two works discussed above, CLEAN assigns a block size for a region of memory statically, at compile time or when a region of memory is allocated. The arguments for this decision are twofold. First, the improvement reported by the works of Dubnicki and LeBlanc, and Johnson, Merten and Hwu may perhaps not justify the cost of a dynamic solution. Second, spatial locality has perhaps a more static, input-independent nature compared to temporal locality. Temporal locality may be very dependent on input size; for a small enough

1. I compute the average speedup by dividing the sum of all running times using the best static line size over the sum of all running times using the variable line size technique.

input size the system will observe good temporal locality as all the data remain cached. But as the input size grows, data become less and less likely to remain cached due to conflicts among data references. Blocking algorithms help mitigate these destructive interferences [LRW91]. In contrast, spatial locality may have a substantial component that is directly associated with the memory layout of related data. A memory hierarchy may be able to exploit this locality component independently of input size. For instance, users often organize code such that when a function accesses a field of a data structure, that function accesses also most of the other fields.

Kumar and Wilkerson propose spatial footprint, a technique that dynamically records usage of data in a cache line while it is present in the cache [KW98]. The cache keeps that information across cache replacements and uses to fetch data in a cache line selectively. This technique addresses mainly the bandwidth requirements between levels of the memory hierarchy.

Johnson, Kaashoek, and Wallach propose CRL, an all-software DSM system [JKW95]. Like CLEAN, their system does not require compiler, operating system, or special hardware support. Unlike CLEAN however, CRL maintains the consistency of shared-data through the concept of regions that the user must specify.

Sandhu, Gamsa, and Zhou proposed a system very similar to CRL [SGZ93]. Programmers declare “shared regions” the accesses of which must be properly demarcated with the following primitives: *ReadAccess()* and *ReadDone()* or *WriteAccess()* and *WriteDone()*.

The advantage of these approaches is that it improves considerably the portability of their tools since they do not rely on executable editing, which may not be available on a target system and is inherently less portable. The drawback of their approach is that the program correctness is no longer decoupled from its performance. The programmer must worry both about a program's correctness *and* performance simultaneously. CLEAN, on the other hand, does not require the programmer to annotate the program for correctness. CLEAN views annotations as optional to improve the program's performance.

Cid is a variant of C extended with constructs to specify multithreaded, distributed shared-memory programs [Nik94]. Programs declare objects; before these programs can safely access these objects, they must first bring locally with the proper call to read or write objects that might be remote.

Midway is a distributed shared memory system proposed by Bershad and his colleagues [BZS93]. Midway is perhaps the system that is the most similar to CLEAN. Midway supports a memory model called *entry consistency* which guarantees that shared data becomes consistent at a processor only when the processor acquires the synchronization object that guards the data. Because this model offers no guarantee with respect to the consistency of other data not protected by the synchronization object, Midway can reduce the frequency of global communication by exploiting synchronization patterns among processors. Like CLEAN, Midway requires the relationship between data and synchronization object be made explicit only when the programmer wants to improve a program's performance. Unlike CLEAN, Midway does not support direct transfer of a lock from the

releaser to the next acquirer (data must transit through the object's server [FBS89]) and Midway does not provide the ability to prefetch synchronization data.

Wood and his colleagues describe CICO [HLRW92, HLRW93, WCF+93, LCW94], a framework in which a user can annotate a program to improve its performance characteristics. As in CLEAN, annotations in CICO are optional; however while CLEAN is a distributed shared memory system, CICO is only a performance model.

## 1.4 Thesis organization

The organization of the remainder of this thesis is as follows. Chapter 2 describes the experimental methodology employed throughout this thesis. Chapter 3 describes the decomposition of a synchronization period, describes a set of four fundamental optimizing locking mechanisms, and shows how they can reduce different parts of the synchronization period. Chapter 3 also reviews the synchronization primitives and discusses how each of them uses a different set of the four mechanisms; and presents the performance results. Chapter 4 discusses the six hardware mechanisms required to support the four locking mechanisms and describes implementation alternatives suitable for current or future systems. Chapter 4 also compares the relative costs of these implementation alternatives. Chapter 5 motivates, describes, and measures the implementation of a programming model that makes use of the results on efficient synchronization support discussed in earlier chapters. Finally, Chapter 6 presents the conclusions and suggests future directions to explore.

# Chapter 2

# Experimental methodology

This chapter details the experimental methodology, illustrated in Figure 2.1, that I employ for the all experiments reported in this thesis. These experiments consist of running parallel programs either on a simulation platform or on an actual parallel machine, and collecting metrics of interest.

All the parallel programs examined in this dissertation are written in C [KR88], extended with annotations that add parallel shared-memory semantics to C. In all cases, the programs use the annotations proposed by Boyle and his colleagues [BBD+87].[1] The parallel programs studied here take one of two forms. Either they are microbenchmarks used to evaluate a particular aspect of the system under study; or they are complete applications more representative of programs that a real environment may run. A detailed description of the two microbenchmarks used in my experiments appears in Section 2.1; descriptions of the six applications analyzed in this study appear in Section 2.2.

1. Many publications refer to these annotations as the `parmacs` macros.

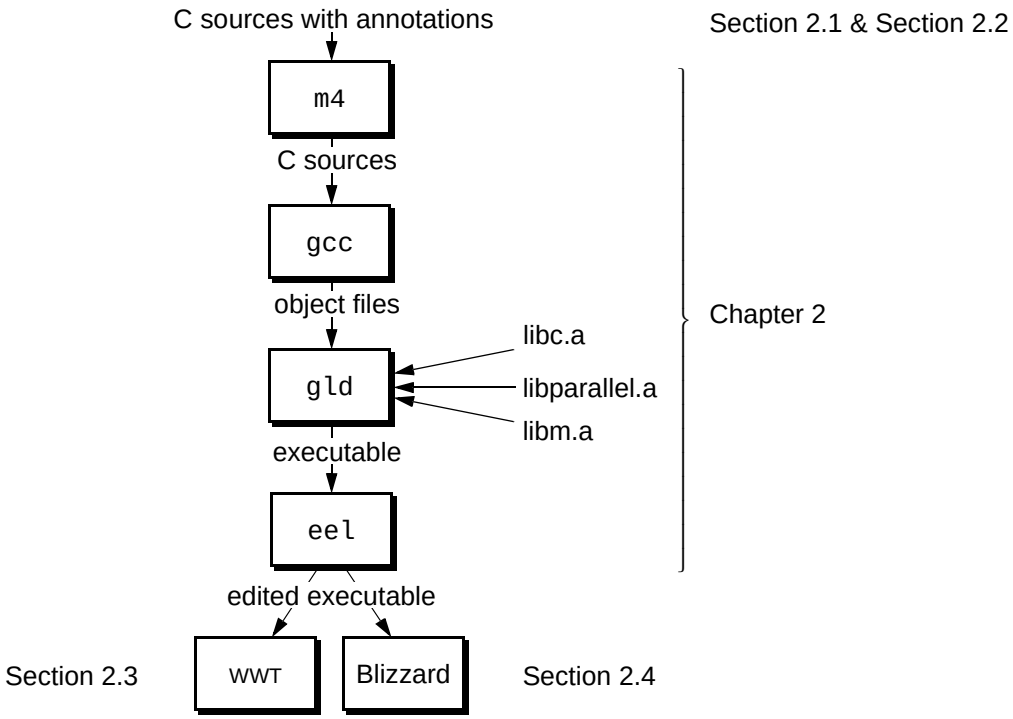



**Figure 2.1 Experimental structure.** This flowchart illustrates the relationship among the tools used to produce the results presented in this study. A preprocessor (`m4`) transforms parallel programs into C source files replacing shared-memory parallel annotations with appropriate functions calls. The compiler (`gcc`) produces objects files that are linked together (`gld`) with the necessary libraries. Finally, the executables are edited and run on either the Wisconsin Wind Tunnel simulator (WWT) or Blizzard, a distributed shared-memory system. A more detailed description of each step appears in the indicated sections.

As shown in Figure 2.1, a succession of tools transforms parallel programs into a form suitable for simulation or actual execution. First, a preprocessor (`m4`) transforms parallel programs into plain C sources substituting the parallel shared-memory annotations into C statements or suitable library calls. Next, version 2.6 of the GNU C compiler (`gcc`) produces object files that the GNU linker (`gld`) links together to create executables. All the programs are compiled with the maximum level of optimization (`-O3`). The resulting executables cannot run

directly on the two platforms (a parallel simulator and a cluster of workstations) that I employ for all my experiments. An additional step, after linkage, rewrites the executables using a binary editor (`eel`). To maintain causality among the nodes in the parallel simulator, the binary editor inserts tests in the executables and forces synchronization among the nodes, if necessary. To support the shared-memory abstraction on the cluster of workstations, the binary editor inserts checks before each load and store instructions in the executables and invokes a user-supplied cache-coherence protocol if the accessed datum is not available locally or is not in the proper state.

Once the edited executables are produced, the simulator or the cluster of workstations can execute them. Section 2.3 describes the parallel simulator (WWT) running on a Thinking Machines' CM-5 [LAD+92] used to study the performance of synchronization primitives. Section 2.4 describes the distributed shared-memory system (Blizzard) running on a cluster of SPARC-based workstations used to evaluate the implementation of various synchronous primitives on an actual hardware platform. To conclude this chapter I present, in Section 2.5, a brief characterization of the studied parallel applications.

## 2.1 Microbenchmarks

### 2.1.1 Standard microbenchmark

Figure 2.2 shows the principal microbenchmark used in this dissertation. This microbenchmark models an application that repeatedly accesses a small critical section in a loop. Once the processor obtains the lock, it waits for a constant

amount of time (*delay* cycles) simulating time spent in the critical section by the application. After the release, the releasing processor waits for a random amount of time. The random waiting time is selected from a discrete uniform distribution over the interval [0, 10×*delay*] with a mean of 5×*delay*. This random delay simulates the application's computation between lock accesses. The statistical nature of the delay makes it improbable, though not impossible, for a single node to obtain the lock twice in succession. I typically run this benchmark several times, each time with a different number of nodes, which varies the contention level to the lock. I report the completion time of all loop iterations of the slowest node. As the number of nodes increases, so does the contention for the lock, and eventually

```
inputs:
    P : int                          [Number of processors.]
    N : int                          [Number of iterations.]
    delay : int                      [Time holding the lock.]

variables:
    counter : int
    lock : int

counter ← ⌊N/P⌋                      [Assign iterations evenly among processors.]
while (counter > 0)
    acquire(lock)
    wait(delay)                      [Wait a fixed amount of time.]
    release(lock)
    wait(random() mod (10 × delay))  [Wait on average five times delay.]
    counter ← counter − 1
end
```

**Figure 2.2 Principal microbenchmark.** This microbenchmark models an application that accesses a critical section repeatedly. This code assumes that the lock is declared in the regular shared address space and that the lock sits by itself in a cache line (to avoid false sharing).

the reduction in the loop execution time stops (and in some cases reverses) when the serialized execution of the critical section dominates the loop completion time.

Note that in my experiments, I do not run the microbenchmark long enough to ensure that the stream of random numbers look “random enough.” Recall, however, that the purpose of the random delay is merely to prevent a single processor from being able to acquire and release a lock repeatedly without intervening remote accesses. I do, however, make sure that the mean waiting time is 5×*delay*.

This methodology is similar to that used by both Anderson [And90], and Lim and Agarwal [LA94] to measure synchronization inefficiencies. However, instead of reporting the microbenchmark completion time directly; they graph the difference between their measurements and an ideal execution time of the microbenchmark assuming a perfect synchronization primitive. In effect, this difference measures the inefficiency of a synchronization primitive for a given contention level. When there is no contention, the inefficiency corresponds to the time spent acquiring and release a lock; when there is contention the inefficiency corresponds to the time to transfer the lock from one node to another. They derive the ideal execution time through simulation from the critical section execution time and the mean delay between lock accesses.

### 2.1.2 Extended microbenchmark

I extend the microbenchmark described in the previous section to explore the impact of collocation on synchronization inefficiencies. Figure 2.3 illustrates the extension. Instead of waiting a fixed amount of time in the critical section, proces-

sors write a globally shared variable modeling accesses to resources protected by a lock. If the programmer does not collocate lock and variable, the program may incur the additional delay of having to obtain an exclusive copy of the variable once in the critical section. Collocation eliminates (in this case) this additional delay completely.

```
inputs:
    P : int                          [Number of processors.]
    N : int                          [Number of iterations.]
    delay : int                      [Set to the typical write miss latency.]

variables:
    counter : int
    monitor : (
        lock : int,
        pad : int[...],              [Remove padding if collocation is desired.]
        variable : int)

counter ← ⌊N/P⌋                      [Assign iterations evenly among processors.]
while (counter > 0)
    acquire(monitor.lock)
    monitor.variable ← 1             [Set variable protected by lock to a value.]
    release(monitor.lock)
    wait(random() mod (10 × delay)) [Wait on average five times delay.]
    counter ← counter – 1
end
```

**Figure 2.3 Extended microbenchmark.** This code is an extension of the microbenchmark that appears in Figure 2.2 to analyze the benefits of collocation. This code assumes that the lock sits in a cache line by itself, unless collocation of the lock with the associated variable is desired. In the latter case, the lock and the variable sit in the same cache line by themselves. In this example, collocation is prevented by introducing enough space between the declaration of the lock and its associated variable.

## 2.2 Shared-memory applications

The benchmark applications used for all experiments performed in this thesis are Barnes, Mp3d, Ocean, Pthor, Raytrace, and Water-Nsq. Mp3d and Pthor are drawn from the SPLASH suite [SWG92], while Raytrace and Water-Nsq are from the SPLASH-2 suites [WOT+95]. Barnes and Ocean are available both from SPLASH and the newer SPLASH-2 benchmark suite; I use the SPLASH version of these benchmarks in Chapter 3, while I use the newer versions in Chapter 4 and Chapter 5. The main reason I use the newer versions of these benchmarks for the later chapters is that I benefit from benchmarks already ported to the platform on which I run experiments. The locking structures are similar in both versions of these benchmarks, and I do not directly compare the results of Chapter 3 with the results of Chapter 4 and Chapter 5. I list the problems that the six benchmarks solve and the inputs that I use in Table 2.1. I pad data in each benchmark, where necessary, to eliminate false sharing [GW88].

**Table 2.1 Benchmarks.**

| BENCHMARK | TYPE OF SIMULATION | INPUT | SOURCE |
|---|---|---|---|
| Barnes | *N*-body | 2,048 bodies, 11 iter. | SPLASH/SPLASH-2 |
| Mp3d | Hypersonic | 24,000 mols, 25 iter. | SPLASH |
| Ocean | Hydrodynamic | 98x98, 2 days | SPLASH/SPLASH-2 |
| Pthor | Digital circuit | `risc`, 1,000 timesteps | SPLASH |
| Raytrace | 3-D rendering | `teapot` | SPLASH-2 |
| Water-Nsq | N-body molecular dynamics | 512 mols, 3 iter. | SPLASH-2 |

Detailed descriptions of all the benchmark applications appear elsewhere [SWG92, WOT+95]. In this section, I only present a short description of each benchmark that is sufficient to understand the application's locking structure and to understand how I rearrange the data layout to study the benefits of collocation. For convenience, I divide each benchmark's description into four parts: (1) sequential algorithm, (2) exploiting parallelism, (3) locking structure and collocation strategy, and (4) other modifications. In the first part (sequential algorithm), I give a short description of each application and an overview of the application's algorithm. The second part (exploiting parallelism) summarizes the parallelization strategy employed in each application. The third part (locking structure and collocation strategy) describes the locking structure and discusses the data restructuring applied to each application to take advantage of the collocation optimization (the original versions of each application do not collocate locks and data protected by these locks). Finally, in the last part (other modifications), I discuss, if applicable, changes that improve the base algorithms (and thus benefit all applications' runs).

Maintaining the original algorithm and locking structure, I always attempt to optimize every benchmark such that each performs well for all experiments I run. In particular, when comparing two locking alternatives, I take care to ensure that the base case performs well for its given synchronization structure. (In a recent paper, Jiang, Shan, and Singh show how to restructure the synchronization structure of certain SPLASH-2 benchmarks to improve their performance [JSS97]; I do not use these restructured codes in this thesis.)

### 2.2.1 Barnes

Barnes simulates the interaction of a system of bodies. This benchmark appears in both the original SPLASH benchmark suite [SWG92] and the SPLASH-2 suite [WOT+95]. I delay the discussion of the differences between these two versions until after the description of the parallel algorithm.

**Sequential algorithm.** This benchmark tracks the evolution of an *N*-body system under the influence of gravitational forces. Specifically, Barnes discretizes time and for each discrete time step (an iteration) computes new positions of the bodies in the system. To avoid computing all $O(N^2)$ interactions among the bodies, Barnes approximates the force exerted by a sufficiently distant cluster of bodies by the force resulting from the cluster's center of mass. This approximation reduces the number of computed interactions to $O(N\log N)$ or even $O(N)$ depending on the distribution of bodies in the system. Central to this approximation technique is the octree data structure. Leaf nodes in the tree represent the actual bodies; while the other nodes, the cells, represent a portion of the three-dimensional space holding the cells' children. A cell bisects the parent cell in all three dimensions. When computing the forces exerted by other bodies, Barnes walks down the tree in breadth-first-search fashion and stops whenever (1) it reaches a leaf or (2) the considered node's center of mass is sufficiently far away, whichever comes first.

**Exploiting parallelism.** Two phases in each iteration consume almost all of Barnes' execution time. The first phase loads bodies in the tree and the second phase computes the interactions. The parallelization strategy in Barnes is owner

based: each process is responsible for a fraction of the bodies in the system. In the tree-building phase, each process loads its bodies in the octree using locks to ensure atomic updates of the cell nodes. In the interaction computation phase, each process computes the forces exerted by other bodies for each body that they own; because a process writes only the bodies it owns, this phase does not require mutual exclusion.

The newer version of Barnes differs from the original version in one important aspect: it allows multiple bodies per leaf in the tree-structured representation of physical space. This change does not affect the overall algorithm. However, allowing more bodies per leaf reduces the height of tree, speeding up the tree-building phase of the algorithm [HS95]. But a shorter tree also slows down the interaction computation phase of the program since the presence of multiple bodies per leaf forces more interaction to be considered. Holt and Singh [HS95] observe that 10 bodies per leaf is the optimum between tree building and interaction computation: the interaction phase slows down; nevertheless the overall execution time speeds up.

**Locking structure and collocation strategy.** The original version of Barnes stores the locks associated with the cells in a separate array. However, to explore the benefits of collocation, I instead store the locks in the cells directly. The benefits of this new layout are twofold. First, it removes some unnecessary contention introduced by the fixed size lock array. Since the number of elements in the array is less than the number of bodies in a typical simulation input, multiple bodies will map to the same lock in the array creating artificial contention. Second the

new layout makes collocation possible. Indeed, each lock is now located near the data it is protecting. The collocation strategy is as follows. The lock and the pointers connecting a cell to its children are placed in the same cache block; the remaining fields of the cell data structure sit in a separate cache block.

### 2.2.2 Mp3d

Mp3d is Monte Carlo simulation of rarefied fluid flow, it is a member of the original SPLASH benchmark suite [SWG92].

**Sequential algorithm.** This benchmark simulates the hypersonic flow of particles at extremely low densities. Succinctly, Mp3d simulates the trajectories of particles through an active space and adjusts the velocities of the particles based on collisions with the boundaries (such as the wind tunnel walls) and other particles. After the system reaches steady-state, statistical analysis of the trajectory data produces an estimated flow field for the studied configuration. The significance of the algorithm implemented in Mp3d is that it reduces the $N^2$ problem of finding collision partners to order $N$. Mp3d finds collision partners efficiently by representing the active space as an array of three-dimensional unit-sized cells. Only particles present in the same cell at the same time are eligible for collision consideration. If the application finds an eligible pair, it uses a probabilistic test to decide whether a collision actually occurs.

**Exploiting parallelism.** The parallel version of Mp3d allocates work to each process through a static assignment of the simulated particles. Each simulated step consists of a move phase and a collide phase for each particle that the pro-

cess owns. The move phase computes the particle's new position based both on its current position and velocity, and its interaction with boundaries. The collision phase determines if the particle just moved collides with another particle; and if so, adjusts the velocities of both particles. Data sharing occurs during collisions and through accesses to the unit-sized space cells. During a collision, a process may have to update the position and velocity of a particle owned by another process. Also, each space cell maintains a count of the particle population currently present in that cell. Therefore, each time a process moves a particle, it may have to update the population count of some space cells if that particle passes from one cell to another. These data accesses to particles and space cells may lead to race conditions that optional locks will eliminate at some performance cost. In this thesis, I study Mp3d compiled with these locks. The presence of locks will typically slow down the program's execution but eliminates the data races and allows repeatability of results.

**Locking structure and collocation strategy.** Mp3d eliminates race conditions while accessing particles and space cells with optional locks associated with each space cell. Since processes update particle information owned by other processors only during a collision and a collision can only occur if two particles are present in the same cell, the space cell locks are programmed to ensure mutual exclusion for both particle and space cell accesses. For experiments with collocation, I place all of the space cell data structure in the same cache line as the corresponding lock.

### 2.2.3 Ocean

Ocean simulates the eddy currents in an ocean basin and appears both in SPLASH [SWG92] and SPLASH-2 [WOT+95]. The newer version of Ocean partitions data differently to improve the communication to computation ratio, improves data locality by allocating contiguously related data, and uses an improved equation solver.

**Sequential algorithm.** This algorithm studies the role of eddy and boundary current in influencing large-scale ocean movements. Specifically, the algorithm simulates a cuboidal basin using a discretized circulation model that takes into account wind stress from atmospheric effects and the friction with ocean floor and walls. The algorithm performs the simulation for many time-steps until the eddies and mean ocean flow attain a mutual balance.

The work performed every time-step essentially involves setting up and solving a set of spatial partial differential equations. For this purpose, the algorithm discretizes the continuous functions by second-order finite-differencing, sets up the resulting difference equations on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin, and solves these equations using the Gauß-Seidel with Successive Over Relaxation (SOR) iterative method (the SPLASH-2 version uses a red-black Gauß-Seidel multigrid solver).

**Exploiting parallelism.** Grid-based algorithms afford much parallelism that applications can easily exploit. The referencing behavior of Ocean is regular and input independent, which the parallel algorithm takes advantage of by permanently assigning grid tasks to processes. Each task performs the computational

steps on the section of the grids that it owns, regularly communicating with other processes. Communication among processes takes the following forms: (1) barrier synchronization to preserve dependences between certain computations; (2) near-neighbor communication while computing the Jacobians and Laplacians (9-point and 5-point stencils, respectively); and (3) updates of a counter by all processes for every SOR iteration to determine convergence.

**Locking structure and collocation strategy.** The program uses locks in two situations. A first lock ensures that each process updates a global sum correctly in order to compute a matrix integral. A second lock helps determine when the SOR iterations have converged. In both cases the algorithm uses a simple lock rather than a tree of locks to perform the reduction. When appropriate I collocate the first lock with the global sum and I collocate the second lock with the counter, which each process increments when it has converged.

**Other modifications.** I increased the array storage of the SPLASH version of Ocean (from 128 to 131 elements in each dimension, slightly increasing the size of the working arrays) to create arrays of prime size, thus reducing cache conflicts among elements in the arrays [LRW91].

### 2.2.4 Pthor

Pthor is a parallel digital circuit simulator drawn from SPLASH [SWG92].

**Sequential algorithm.** This benchmark, an event-driven simulator, simulates the behavior of a digital circuit over time. A circuit consists of elements attached together through wires or nodes. Each element takes signals as inputs which

combined with the associated internal state determine the values driven on the output ports. The nodes, then, transmit the values from the output ports of an element to the input ports of other elements. Elements can be as simple as an AND-gate and as complex as a complete CPU. The value carried by an interconnecting node is one of high, low, undefined, or floating. Each change in a node value corresponds to an event associated with a time at which the change should occur. The algorithm chronologically stores these events in a global event queue. The algorithm takes, then, the oldest event in the queue and drives the inputs ports attached to the corresponding node according to the new value that this node is taking on. As the effect of the change propagates in the circuit it generates new events which are inserted in the queue. The algorithm stops when either the event queue is empty or the algorithm has simulated the circuit for the user-specified amount of time-steps.

**Exploiting parallelism.** Pthor exploits the parallelism available in the processing of the events stored in the event queue. Pthor uses a variant of the Chandy-Misra distributed-time algorithm [CM81] to simulate the circuit efficiently. The program distributes the event processing among the processes by assigning each of them an event queue. In effect, each process may have a different notion of what constitutes the current time (the time of the event at the head of its queue) that may lead to a deadlock (when no processor has any event left to process). The algorithm has a provision to resolve the deadlock and resume the normal distributed processing of events.

**Locking structure and collocation strategy.** The principal locks in Pthor are the locks associated with the event queues and the locks associated with the elements of the simulated circuit. The collocation strategy is as follows. Each event queue has two pointers: one points to the head of the queue, while the other one points to the tail. To avoid unnecessary synchronization, Pthor checks the status of the queue without acquiring a lock first. This check is performed by accessing the head of the queue. Since access to the head is not always performed within a critical section, I do not collocate it with the queue lock. On the other hand, tail is always updated within a critical section, therefore I collocate this variable. Also, I collocate the fields of the element data structure with its corresponding lock.

**Other modifications.** I improve the spatial locality of the circuit data structure by rearranging some of its fields. After initialization, the access patterns to these fields fall into two categories: read-only (e.g., the number of output ports of an element) or migratory (e.g., the state of the element). By placing read-only fields together in the same cache line at the exclusion of migratory fields, Pthor can take full advantage of the automatic replication of read-only data across the caches of the system. By collocating migratory fields, Pthor can take advantage of the prefetching property of large cache lines.

### 2.2.5 Raytrace

Raytrace, a ray tracer, is a member of the SPLASH-2 benchmark suite [WOT+95].

**Sequential algorithm.** This benchmark renders a three-dimensional scene using ray tracing. It generates primary rays starting from the viewpoint, through the image plane, to the scene. When a ray encounters an object in the scene, the algorithm reflects that ray towards each light source to determine if the ray is shielded from that light source, and if not the algorithm computes the contribution of the unshielded light source. Objects also spawn new rays as the primary ray either reflects or refracts (as appropriate) from or through, respectively, the surface of these objects. The algorithm continues to spawn new rays recursively, forming a tree of rays, until either a ray leaves the scene or a new ray would exceed some user-defined threshold (such as the maximum number of levels allowed in the ray tree).

**Exploiting parallelism.** The algorithm affords much parallelism across rays. The parallel version of Raytrace distributes work among the processes through a distributed task queue. The parallel program ensures the correct operation on the queue with locks. The program provides efficient access to the scene description through (1) hierarchical uniform grid to traverse scene data quickly, (2) round-robin distribution of the scene data across the nodes in the system to balance load on the network and the memory modules, and (3) replication of data in the caches (the scene data is read-only).

**Locking structure and collocation strategy.** Raytrace uses locks in two important circumstances. In the first one, a lock protects access to a counter used to assign a unique identifier to each newly spawned ray. The critical section consists only of fetching the counter, adding one to it, and storing it back to memory.

Contention to that lock is very high. Another set of locks (one lock per queue) ensures the correct operation on the distributed task queue (one queue per processor). Contention to these locks is typically fairly low, unless the number of participating processors approaches the number of rays created. I perform collocation as follows: I place the unique identifier counter in the same cache line as its associated lock and I place each queue lock in the same cache line as the address pointing to the first enqueued task.

### 2.2.6 Water-Nsq

This *N*-body molecular dynamics application is a member of the SPLASH-2 benchmark suite [WOT+95].

**Sequential algorithm.** This benchmark computes the forces and potential of water molecules in a cubical box to predict a variety of static and dynamic properties of liquid water. For a user-specified number of time-steps, this program estimates the forces each molecule exerts on all others according to the Newtonian equations of motion. Water-Nsq avoids computing all $N^2$ interactions by eliminating from consideration molecules outside of a sphere centered at the examined molecule and of a radius corresponding to half of the box length.

After some initialization and one-time computations, each time-step consists of five phases: (1) calculating the predicted values of atomic variables; (2) computing intra-molecular forces for all atoms; (3) computing the inter-molecular forces; (4) calculating the corrected values of variables from the predicted values and computed forces; and (5) computing the total kinetic energy of the system. The third

task (computing the inter-molecular forces) accounts for the most of the execution time: its time complexity is $O(N^2)$ while all the other tasks have a time complexity of $O(N)$.

**Exploiting parallelism.** Water-Nsq affords a lot of parallelism, both across phases and within phases. This parallel version of Water-Nsq exploits mostly the parallelism available within a phase; it exploits the inter-phase parallelism to a limited extent to avoid some synchronization between phases. To exploit locality, Water-Nsq both assigns statically each processor an even fraction of the molecules and stores the molecules assigned to the same processor next to each other. Communication among processors occurs during the second (intra-molecular computation) and third (inter-molecular computation) phases. Communication in the second phase consists only of adding scalars into a global sum; locks ensure that the processors correctly update that sum. Communication also occurs in the inter-molecular computation, where processors read positions of the interacting molecules, compute the forces, and update the forces of both molecules. A lock per molecule ensures the atomicity of the force updates.

**Locking structure and collocation strategy.** In the intra-molecular computation, each processor updates a global sum protected by a lock to ensure correct operation. For the collocation experiments, I place both the lock and the accumulator in the same cache line. In the original Water-Nsq, a separate array of locks ensures proper updates of the forces associated with each molecule. To enable collocation of these locks with each molecule's forces require a reorganization of the layout of molecules in memory. Water-Nsq stores the positions of each molecule as

well as their first five moments (velocities, accelerations, etc.) and the forces consecutively in an array. To enable collocation, I store, instead, all moments associated with each molecule as fields of a structure. With this new layout, it is straight-forward to collocate a lock with each field that stores the forces.

## 2.3 Simulation environment

The main simulation platform is the Wisconsin Wind Tunnel (WWT) [RHL+93], which uses a 32-processor Thinking Machines CM-5 [LAD+92] as its host machine. WWT executes SPARC binaries in native mode on the CM-5, only trapping into the simulator upon a cache miss. WWT assumes fixed execution time for the instructions (the actual values correspond to the instruction delays listed in the CY701 SPARC user's guide [Cyp90]). WWT makes some assumptions about the target system to simplify simulation—it assumes both a perfect instruction cache and that stack accesses always hit in the data cache.

The default WWT network model assumes a fully connected point-to-point target network, in which messages take a constant number of cycles for a one-way network traversal. A large enough constant latency provides sufficient lookahead for efficient parallel simulation, as host nodes stop and synchronize only once every *C* cycles, where *C* is the constant network latency. Using a small *C* (or variable-length messages) reduces the node lookahead, which causes severe increases in simulation time [BW95].

Although I model contention at the target node interfaces, memory, and memory directories, using a constant network latency ignores contention in the net-

work itself. To account for network contention, I use an analytical model [SGV92] (which takes the network load as a parameter) to derive a different average network latency for each benchmark. I estimate the aggregate network load from the traffic statistics of previous simulations and their total execution times. Since the network latency affects execution time and therefore aggregate load, I iterate this estimation until the difference between the network latency constant and the average value produced by the model converge to within one cycle (the final latencies for the benchmarks ranged from 85 to 112 processor cycles for current technology parameters, and from 165 to 188 processor cycles for future technology parameters).

To validate this process, I use a detailed, event-driven network simulator (based on the original WWT network simulator [BW95]) that accurately simulates message buffering, message retransmission, and flow control [BG95]. The implementation serializes the network simulation at a central host node, making simulation performance suffer by roughly a factor of 15.

The target network used for the validation is an 8×4 mesh of rings. The target network routes the requests in increasing dimension order and responses in decreasing order. The internal details of the simulated network correspond closely to those of the SCI transport layer standard [IEE93]. A message's delay through the network includes staging time at the source and target nodes, parsing and wire delay through each intermediate node, and possibly a delay through an agent queue, if the message switches dimensions. Table 2.2 lists the specific times for these delays, for both current and future networks.

**Table 2.2 Parameter settings.** All delays are in CPU cycles. The network is assumed to have two byte-wide links. Parsing delay accounts for the time spent on the routing decision; wire delay accounts for buffering and multiplexing; agent delay accounts for dimension switching delays; and staging delay accounts for data transfers occurring at the source and target nodes.

| TECHNOLOGY | CURRENT | FUTURE |
|---|---|---|
| NODE PARAMETERS | | |
| Processor speed | 200 MHZ | 500 MHZ |
| Sustained IPC | 1 | 2 |
| Cache access | 3 | 12 |
| Directory access | 10 | 40 |
| NETWORK PARAMETERS | | |
| Network bandwidth | 500 MB/S | 1 GB/S |
| Parse delay | 4 | 6 |
| Wire delay | 3 | 8 |
| Agent delay | 22 | 28 |
| Staging delay | 14 | 28 |

Table 2.3 shows the errors (in terms of target execution time) that the constant latency network model suffers when compared against the detailed network simulation. The two columns of network latencies represent the mean message latency returned by the SCI network simulator and the model, respectively. The error columns show the error in target execution time that I calculate by comparing the constant latency runs against runs that used the SCI network simulator. These network validation runs assume sequential consistency and MCS locks, and the benchmarks use smaller data sets than the other experiments reported in this thesis.

**Table 2.3 Inaccuracies of the constant latency network model.** This table compares the target running time returned by the detailed network simulator against two runs of the constant-latency simulator. In the first run, the simulator uses as the constant network latency the mean computed by the detailed simulator. In the second run, the simulator uses the mean produced by the model.

| BENCHMARK | SIMULATED | | MODELED | |
|---|---|---|---|---|
| | LATENCY | % ERROR | LATENCY | % ERROR |
| Barnes | 93 | –1.18 | 88 | 2.12 |
| Mp3d | 92 | 0.26 | 89 | 1.72 |
| Ocean | 93 | 1.46 | 93 | 3.08 |
| Pthor | 102 | –2.09 | 88 | 4.61 |
| Water-Nsq | 91 | –0.27 | 87 | –0.04 |

Unless stated otherwise, the target systems that I simulate consist of a cache-coherent shared-memory system with 32 nodes. The coherent caches use the Scalable Coherent Interface (SCI) [43] as their base cache-coherence protocol. SCI defines, among other features, an option for efficient support for synchronization (QOLB). Each node in the target system is workstation-like, containing a processor, a 1-Mbyte four-way set-associative cache memory with 64-byte lines, a 64-entry transaction queue, a network interface, and an even fraction of the distributed, globally-shared memory with the associated directory entries. The transaction queue is similar to a functionally extended write buffer. It supports the following asynchronous operations: writes, prefetches, sharing-list invalidations, and cache line flushes caused by replacement.

A complete description of the system parameters and their associated timings follows: a 64-byte line size, consistent with the SCI standard, a one cycle hit time for the caches, and 1 one cycle per 32-bit word fill time. Cache replacements are selected based on which line least recently missed or write-faulted (this is a constraint imposed by WWT). Memory loads always block, and main memory has an invariant of one cycle per 32-bit word fill time. WWT allocates private target pages locally, and distributes shared target pages to the target nodes round-robin. The simulated memory system supports release consistency [GLL+90].

Simulation results for two types of systems are presented in this thesis: target systems using technology current today and target systems using future technology (my estimates are for approximately five years in the future). Table 2.2 lists the system parameters for both technology levels (*current* and *future*). Note that the future processor is effectively five times as fast in terms of instructions executed per unit time.

I perform experiments with different sets of assumptions to explore the impact of technology advances. In particular, I wish to discover whether system parameters that change due to technological improvement will qualitatively change the results with respect to the performance benefits of efficient synchronization.

## 2.4 Experimental platform: COW

COW, a cluster of workstations, consists of 16 unmodified dual processor Sun SPARCstation 20s, each with two 66-MHz HyperSPARC processors [ROS93] and a Myricom Myrinet interface [BCF+95], running the manufacturers' Solaris release

5.4 operating system. The network topology is a regular tree of degree six. A tree depth of two is sufficient to connect up to 36 nodes. The router at the top level adds about 0.5μs latency to messages that must switch sub-trees [BCF+95].

The workstations use the Blizzard run-time system [SFL+94] to support the shared memory abstraction. Blizzard is an implementation of the Tempest interface [RLW94].

The detection of message arrival is achieved through polling. A binary rewriting tool [LS95] automatically inserts polling instructions and checks before each shared-memory access in the parallel program. By default Blizzard performs polling through an uncacheable read access to a memory-mapped status register. This method of polling wastes memory bus bandwidth when the incoming message queue is empty. To reduce bus contention Blizzard can exploit the fact that the memory bus on the Sun SPARCstation supports coherent memory transactions. The polling code in all the experiments performed in this study checks the status of the network interface through accesses to a cacheable memory-mapped location. The network interface updates this cacheable location using its DMA interface [MFHW96]. This optimization lets the polling complete most of the time without requesting the bus, reducing traffic and bus load. The drawback of this optimization is that the polling latency when there are messages waiting in the queue is slightly higher than if it had been performed using the uncacheable access (an increase of about 1μs).

## 2.5 Application characterization

### 2.5.1 Working sets

Following the methodology proposed by Rothberg, Singh, and Gupta [RSG93], I measure the size of the important working sets for each of the benchmarks used in this study. To illustrate the important working sets of each benchmark, I plot in Figure 2.4 the total cache misses that a program generates for different cache sizes. On the horizontal axis, I vary the cache sizes logarithmically in increments of integral power of twos; on the vertical axis I plot the total cache misses. I count the total cache misses from simulations that I run on the WWT assuming a sequentially consistent memory system and QOLB with all the synchronization mechanisms enabled. I do not expect the memory model or the synchronization primitive employed for these measurements to have a substantial impact on the sizes of working sets.

Simulation environments, owing to the constrained speed at which they can simulate the execution a benchmark, limit the problem sizes that are reasonable to run. The problem sizes that I employ in this study are much smaller than the data set that a real system is expected to run. I perform all simulation experiments described in Chapter 3 assuming two sizes of caches. Most of the measurements assume a 1MB cache, which is a reasonable size for current technology. For the data sets used, a cache of this size enables me to evaluate the performance of locking mechanisms and synchronization primitives without experiencing finite cache effects. However, assuming a large cache, benchmarks with small data sets

may tend to overemphasize the benefits of locking optimizations. Thus, I rerun some of the experiments with a smaller cache size that is more realistic for each input used. To guide my choices of smaller cache sizes I use the working set analyses presented in this section. The initial cache size choice (1MB) is always large enough to fit the largest working set (WS2) for all applications. I then repeat some key experiments with the smallest cache size that will fit the second largest working set (WS1).

Based on the methodology and results by Rothberg, Singh, and Gupta [RSG93] and the SPLASH reports [SWG92, WOT+95], Table 2.4 summarizes the important working sets of each of the six analyzed benchmarks. In the following paragraphs, I give a more detailed description of the important working sets of each applications.

**Barnes.** The first important working set (WS1) corresponds to the body and the portion of the tree that Barnes needs to traverse in order to compute the forces acting on that body. The size of that sub-tree is proportional to the logarithm of the number of processors and inversely proportional to the square of a parameter that controls the accuracy of the simulation. The second working set (WS2) corresponds to the maximum of (1) the amount of data in a processor's partition and (2) the amount of data to compute all interactions for the bodies that the processor owns [RSG93]. For the choice of parameters used here, the size of WS1 and WS2 are 64KB and 128KB, respectively.

**Mp3d.** This benchmark has an unstructured data access behavior; nevertheless, a first knee in the curve is fairly well defined at size 4KB and the curve reaches an asymptote at size 512KB.

**Ocean.** The first working set corresponds to the point where an entire column of data fits in the cache (98 elements). The next working set corresponds to the partition of the data set assigned to one processor. The sizes of WS1 and WS2 are 2KB and 128KB respectively.

**Pthor.** The first working set (WS1) corresponds to an element and the nodes that drives it and are driven by it. The size of the second working set (WS2) corresponds to the amount of data required to compute the behavior of the elements assigned to a processor. The respective sizes of the working sets are 2KB and 64KB.

**Raytrace.** The working sets of Raytrace are not well defined due the unstructured nature of the computation. I select the first working set based on the first sharp drop in caches misses (2KB) and select the second working set when the total misses almost reach an asymptote (128KB).

**Water-Nsq.** The first working set (WS1) corresponds to private data. The second working set (WS2) corresponds to the partition of the data set assigned to one processor. The sizes of these working sets are 2KB and 64KB respectively.

**Table 2.4 Important working sets and their sizes for each benchmark.**

| | WORKING SET 1 | | WORKING SET 2 | |
|---|---|---|---|---|
| BENCHMARK | DESCRIPTION | SIZE | DESCRIPTION | SIZE |
| Barnes | Tree data for 1 body | 64KB | Partition of DS | 128KB |
| Mp3d | Unstructured | 4KB | Partition of DS | 512KB |
| Ocean | A few columns | 2KB | Partition of DS | 128KB |
| Pthor | A circuit element | 2KB | Unstructured | 64KB |
| Raytrace | Unstructured | 2KB | Unstructured | 128KB |
| Water-Nsq | Private Data | 2KB | Partition of DS | 64KB |

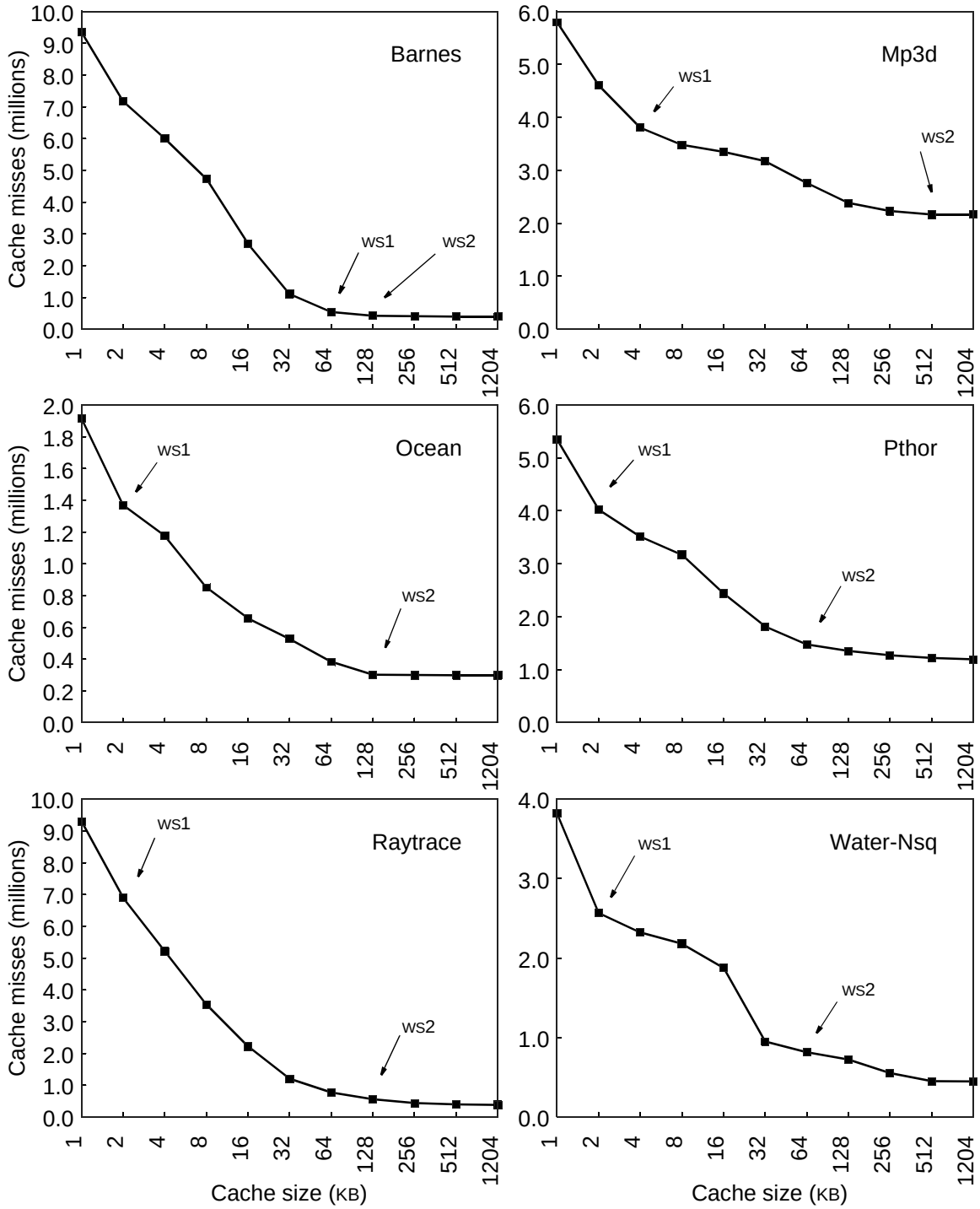



**Figure 2.4 Total cache misses versus cache size.** The cache misses assumes 32 processors with 64 byte line sizes and 4-way associative caches. WS0, WS1 and WS2 refer to the observed working sets.

### 2.5.2 Locking

Table 2.5 summarizes some important static and dynamic statistics about the usage of locks in the six benchmarks. It gives the number of distinct locks that each benchmark uses during execution, the overall number of times that a benchmark accesses these locks, the average number of processor cycles that a benchmark holds a lock (i.e., the average length of the critical section), the probability that a lock access finds a lock busy, and the number of processor cycles that elapse, on average, between two lock accesses. The latter statistic is computed by dividing the benchmark execution time (discounting initialization) by the total number of critical section entries across all 32 processors (the second statistic in the third column).

All the numbers in Table 2.5 assumes a sequentially consistent memory system and uses QOLB with all the synchronization mechanisms enabled.

**Table 2.5 Critical section statistics.**

| | NUMBER OF LOCKS | | CRITICAL SECTION | | AVG # CYCLES BETWEEN LOCK ACCESSES |
|---|---|---|---|---|---|
| BENCHMARK | DISTINCT | ACCESSES | AVG. LENGTH | BUSY | |
| Barnes | 2,052 | 46,724 | 930 | 13% | 1,840 |
| Mp3d | 388 | 1,282,224 | 470 | 16% | 43 |
| Ocean | 4 | 689 | 320 | 96% | 17,936 |
| Pthor | 5,064 | 114,412 | 4,250 | 27% | 674 |
| Raytrace | 4 | 74,413 | 540 | 26% | 495 |
| Water-Nsq | 521 | 100,931 | 320 | 3% | 610 |

# Chapter 3

# Performance of synchronization primitives

## 3.1 Introduction

The performance of a synchronization primitive may contribute to a large extent to the overall performance of a parallel shared-memory program that uses fine-grain locking. Also, it may affect the potential for a program to scale to larger numbers of processors. This chapter characterizes the performance of a number of important synchronization primitives. Also, using a novel decomposition of synchronization primitives into mechanisms, this chapter helps determine the relative merits of those underlying mechanisms.

This chapter begins by defining the synchronization period—a device to reason about the performance and inefficiencies of synchronization primitives. Then, with the help of this device, I discuss the inefficiencies that may occur in synchronization primitives and identify four previously proposed mechanisms that can reduce or eliminate some of these inefficiencies. Next, I discuss a comprehensive list of synchronization primitives proposed to date and show which of these mech-

anisms they incorporate. Finally, using detailed simulations, I evaluate quantitatively the performance of key synchronization primitives. Also, I attempt to evaluate the performance benefits of each of the four proposed locking mechanisms.

Preliminary versions of this chapter appear in the literature [KABG95, KBG97]. Of my co-authors, Aboulenein provided the compiler infrastructure used in the first paper, but which I do not use in this thesis. Burger designed and implemented the detailed network simulator [BG95] that I use to verify some of my experimental assumptions. Burger and Goodman also helped with the organization and writing of the papers. I draw upon these papers for some sections that appear in this chapter.

## 3.2 Synchronization period

Locks provide individual processors with exclusive access to shared data and a critical section of code.[1] This exclusive access is particularly well-suited to the fine-grained nature of many shared-memory parallel programs. Fine-grained programs ideally associate as little data or code as possible with a critical section, minimizing serialized processing, thus maximizing available parallelism. Since the access to critical sections is by definition serialized among processors, large inefficiencies when accessing a contested critical section degrade both parallel performance and potential scalability. To maximize both the performance of fine-

1. Certain applications, most notably database codes and operating systems, distinguish between shared and exclusive locks. A shared lock allows multiple processes to read associated data concurrently; in contrast, an exclusive lock grants a single process the privilege to modify those data. This work focuses only on exclusive locks.

grain parallel applications that use locking and the potential to scale to larger numbers of processors, we must minimize the delays associated with the transfer of exclusively accessed resources.

The act of transferring control of a critical section is complex, and may involve multiple remote transactions. Some complex protocols perform this transfer efficiently, allowing reasonable performance when there is high contention for a lock. The complexity of these protocols, however, causes unnecessary delays when accessing a lock that is free. Conversely, simple locking schemes that can access a free lock quickly may perform poorly in the presence of contention. As a consequence of this trade-off, the literature contains proposals of numerous primitives [AC89, And90, BD86, Cra93, HM93, FG91, Gle91, GVW89, GT90, JHB87, KCK99, LA94, LR90, MCS91b, MLH94, RS84, SSHT93].

To understand where the opportunities for optimization lie, I first decompose the time associated with a complete locking period into three phases: *Transfer*, *Load/Compute*, and *Release*. Together, these phases form a *synchronization period*, which determines the global throughput of synchronization operations and thus determines scalability for codes that rely heavily on locks. Then, I describe four previously proposed mechanisms that locks may incorporate to reduce the time spent in the three phases: (1) *local spinning*, (2) *queue-based locking*, (3) *collocation* (of a lock and data within the same cache line), and (4) *synchronous prefetch*.

## 3.3 Synchronization inefficiencies

From the perspective of an individual processor, the time associated with an access to a critical section consists of the time from which the processor first requests access to the corresponding lock, to the time at which the processor completes the release on that lock. This time period does not directly correlate with global performance, however. Multiple processors contending for entry to the same critical section may overlap the time from the issue of their requests to the first release of the lock. A good analogy to this distinction is the difference between the latency of an individual request to a memory system, and the throughput achievable by pipelined accesses to that same memory system.

To determine how these critical section accesses limit global performance and ultimately scalability, I define the notion of a *synchronization period*. The synchronization period is the length of time between completion of two successive synchronization operations (e.g., two successive releases) on the same variable for the case that the lock has been requested before the first release. The successive synchronization operations may occur on different processors. This synchronization period is the service time that the processor incurs once the previous processor releases the lock. Since access to the critical section is by definition serialized, the synchronization period will place an upper bound on possible throughput (codes that do not access critical sections heavily will see upper bounds on performance from other sources, of course).

I depict the breakdown of a synchronization period in Figure 3.1. The figure shows events to synchronization variable *X*. The first event depicted is the completion of the release of lock *X* by processor A. Several processors are contending to gain access to *X*. I assume that processor B wins the ensuing arbitration. When the lock acquire completes, processor B enters the critical section. Upon finishing the work in the critical section, processor B prepares to release *X*, and eventually completes this operation. The breakdown of a synchronization period consists of three phases:

- *Transfer*: the time at which processor A completes its release of the lock to the time processor B completes its acquire. The release completes when the releasing processor atomically writes the "unlocked" value to the lock. The contending nodes may then re-issue requests (depending on the locking primitive) to obtain the lock. A period of arbitration may ensue. Once the next recipient of the lock is determined, that node must complete its acquire operation, typically on the free lock sent to it.
- *Load/compute*: the time at which processor B completes its lock acquire to the time processor B issues its lock release. Once a processor obtains the lock, it enters the critical section. The processor will most likely have to read some protected data, perform some computation, and write some protected data. Accessing the data to read and write will typically incur some remote accesses.
- *Release*: the time from processor B issuing the lock release to the completion of the lock release. When the processor issues a release operation for the lock, remote accesses may be necessary before that operation can complete. Other

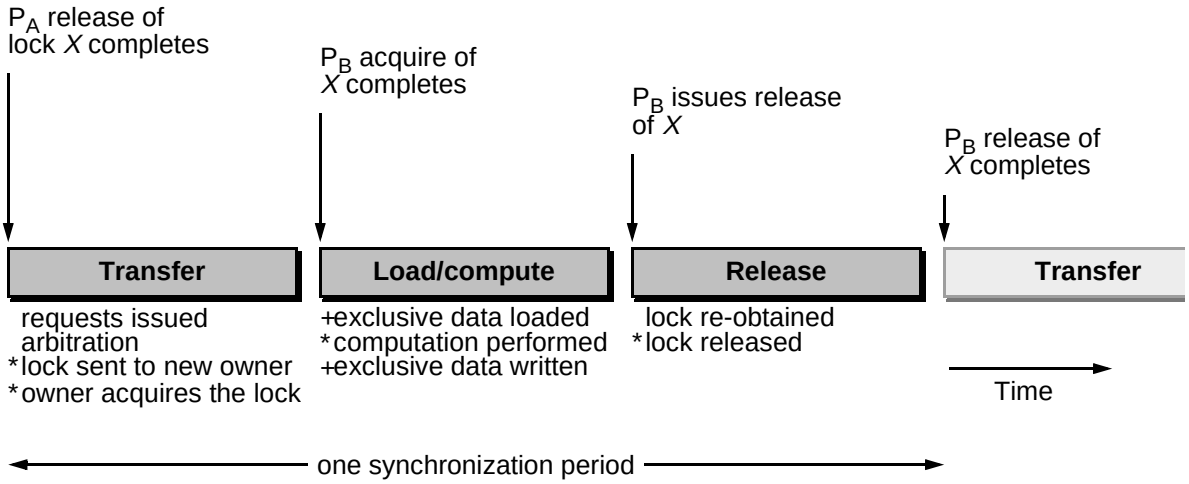



**Figure 3.1 Breakdown of one synchronization period.**

processors may have removed the lock from the releasing processor's cache, for example, or the releasing processor may have to re-obtain write permission for the lock's cache line. If the release operation is implemented with more than one instruction (e.g., MCS), some aggressive memory models [AH90, GLL+90] may allow some overlap between the *Load/compute* and *Release* phases. In particular, consider a memory access performed during the *Release* phase that does not conflict with any other accesses [GLL+90]; it may be permissible to overlap this access with accesses initiated in the *Load/compute* phase.

In addition to illustrating the three phases in Figure 3.1, I also list the components of each phase. The components marked with an asterisk are the only ones that are fundamental, which would be part of a truly minimal synchronization period. The components marked with a "+" are causes of potential inefficiencies, the latencies of which may be partially or entirely hidden. Unmarked components are ripe for elimination through optimization.

## 3.4 Locking mechanisms

I identify a set of four mechanisms that synchronization primitives can incorporate to avoid inefficiencies. In Table 3.1, I show the inefficiencies (from Figure 3.1) reduced by each of these mechanisms. The definitions and explanations of each mechanism are as follows:

- *Local spinning*: allows a requesting node to spin on a local copy of the lock, eliminating unnecessary requests for a held lock. Although local spinning does not directly reduce inefficiencies on the critical path of the synchronization period, it does greatly reduce the load on the network, particularly for longer critical sections.
- *Queue-based locking*: eliminates arbitration inefficiencies and reduces lock transfer time, both in the *Transfer* phase. This mechanism reduces synchronization inefficiencies in the following ways: (1) it creates a queue of waiting requesters, thus performing arbitration when the requests are received and not when the current holder releases the lock; (2) it reduces lock transfer time

**Table 3.1 Inefficiencies addressed by locking mechanisms.**

| | PHASE OF THE SYNCHRONIZATION PERIOD | | | | |
|---|---|---|---|---|---|
| | TRANSFER | | LOAD/COMPUTE | | RELEASE |
| LOCKING MECHANISMS | ARBITRATION | LOCK TRANSFER | DATA READ | DATA WRITE | RE-OBTAIN LOCK |
| Local spinning | | | | | |
| Queue-based locking | ✓ | ✓ | | | ✓ |
| Collocation | | | ✓ | ✓ | |
| Synchronous prefetch | | ✓ | maybe | | |

by restricting communication to be between the releasing node and the acquiring node only (although the number of remote accesses required to perform this transfer will vary among different primitives); and (3) it eliminates the inefficiency of re-obtaining the lock in the *Release* phase, since no other nodes access the lock directly until the holder releases the lock.

- *Collocation*:[1] lets protected data be transferred with the transfer of the lock itself. Since the data arrive with the lock, collocation eliminates read and write inefficiencies in the *Load/compute* phase. The implementations I study in this thesis achieve collocation by coupling a lock and critical data together in the same transfer block (a cache line). If the critical data are larger than one cache line, collocation will only partly reduce the read and write access inefficiencies. If the critical data are determined dynamically, effective collocation is difficult.

- *Synchronous prefetch*: allows a processor to issue a request for a particular lock in advance of its critical section. The memory system will effect the transfer of the lock from the holder to the prefetching requester only when the holder releases the lock. Thus, this mechanism will not impede the current holder's progress in the critical section. If a node prefetches the lock and the holder releases it before the requester reaches its critical section, the requester may be able to hide the lock transfer latency completely.

1. col.lo.ca.tion (n) \.kal-*-'ka-sh*n\: the act or result of placing or arranging together; specif: a noticeable arrangement or conjoining of linguistic elements (as words) [Web65] (*words* in this context are 4-byte quantities of data).

## 3.5 Synchronization primitives

The basic synchronization primitives I discuss in this chapter are test&set, test&test&set, MCS locks, Anderson's locks, Graunke and Thakkar's locks, LH locks, M locks, QOLB, and Lee and Ramachandran locks. Table 3.2 shows which of the mechanisms described in Section 3.4 are incorporated in each of these locks. In Table 3.3, I show the minimum number of remote transactions required for acquiring a lock. Each pair of numbers shown represents the number of transactions required for a DASH-like [LLG+92] and SCI-like [IEE93] protocol, respectively. DASH is the first research prototype of a cache-coherent distributed memory multiprocessor; DASH is a widely studied and well understood system. SCI is the protocol that I use for many of the simulation-based qualitative performance analyses that appear in this thesis. I compute the data that appear in Table 3.3 based on the following assumptions. In cases where the lock is not held (columns 1 and 2), the number of transactions is from issue to completion of the lock acquire. If the lock is held by another node, the number of remote transactions shown is the number from issue of the release by the lock holder to the completion of the acquire by the requester. In the rest of this section, I describe each base primitive and describe each in terms of the mechanisms that it incorporates, as shown in Table 3.2. I also discuss the reactive synchronization schemes [LA94] and other proposals to improve the performance of synchronization operations.

**Table 3.2 Synchronization primitives.** For each synchronization primitive, this table shows which locking mechanisms it incorporates. I deem collocation to be optional, since the programmer may choose not to exercise it.

| | LOCKING MECHANISM | | | |
|---|---|---|---|---|
| SYNCHRONIZATION PRIMITIVE | LOCAL SPINNING | QUEUE-BASED LOCKING | COLLOCATION | SYNCHRONOUS PREFETCH |
| Test&set | no | no | optional | no |
| Test&test&set | yes | no | optional | no |
| MCS, Anderson's, Graunke and Thakkar's, LH, M | yes | yes | partial | no |
| Lee and Ramachandran's[a] | yes | yes | optional | ?[b] |
| QOLB | yes | yes | optional | yes |

a. The capability of the Lee and Ramachandran's lock to support read locks is outside the scope of this study.
b. Lee and Ramachandran's description of their lock [LR90] does not mention prefetching; however nothing inherent to the design of the lock prevents its supporting such an operation.

### 3.5.1 Test&set

Test&set performs an atomic read-modify-write on a memory location. It reads the value contained therein, and unconditionally sets the value to be non-zero. Test&set returns the original value read. It may be implemented with an atomic swap of as little as one bit.

We see in Table 3.3 that the test&set primitive is efficient when a lock is not held; the primitive can immediately load the lock into the processor's cache and lock it. Test&set is less efficient when there is contention for a lock, since the lock's line is shifted from requester to requester in "exclusive" state. When the holder wishes to release the lock, it must re-obtain the lock from the requester that has moved the line into its cache. Concurrently, all requesters continue to send requests for writable copies of the lock. Although this scheme technically

**Table 3.3 Number of remote transfers for acquire.** The numbers in the table represent the minimal number of messages needed to acquire a lock. The counts correspond to messages on the critical path only. I show numbers for several initial lock states and two cache-coherence protocols. Each number on the left assumes a DASH-like protocol [LLJ+92], and each number on the right assumes an SCI-like protocol [IEE93]. I assume that the acquiring node, the releasing node (if applicable), and the directory node are all distinct. In cases where the lock is not held (columns one and two), the number of transactions is from issue to completion of the lock acquire. If another node holds the lock, the number of remote transactions is from issue of the release by the lock holder to the completion of the acquire by the requester.

| | MINIMAL NUMBER OF REMOTE MESSAGES | | | |
|---|---|---|---|---|
| LOCKING PRIMITIVE | LOCK IDLE IN MEMORY | UNLOCKED, CACHED ELSEWHERE | LOCKED, SINGLE CONTESTANT | LOCKED, *N* CONTESTANTS |
| test&set | 2, 2 | 3, 6 | 5, 11 | 5, 11 |
| test&test&set | 4, 2 | 6, 6 | 8, 11 | 8, 9+2×*N* |
| MCS | 2, 2 | 3, 6 | 7, 15 | 5, 9 |
| LH | 2, 2 | 9, 10 | 5, 11 | 5, 11 |
| M | 2, 2 | 3, 6 | 5, 11 | 5, 11 |
| QOLB | 2, 2 | 3, 4 | 1, 1 | 1, 1 |

guarantees that some processor makes forward progress, it does not guarantee fairness, nor does it prevent starvation. Worse, it generates continuous remote transactions from the requesters (if there are more than one), even while the lock is being held. We see from Table 3.2 that the only optimization (of those in the table) that test&set may implement is collocation. Collocation may be effective if requesters rarely attempt to obtain the lock while held. When a request for a held lock occurs, however, the requester and holder will cause the line holding the lock (and collocated data) to ping-pong between their caches, as the holder accesses the data and the requester spins on the lock. The ping-ponging of the block will

stall the holder, increasing the length of its critical section and thus increasing the global synchronization period.

A policy often applied to test&set is exponential backoff, in which after a failure to obtain the lock a requester waits for successively longer periods of time before issuing another request for a lock [And90, AC89]. I implement a backoff scheme closely following the guidelines that appear in the original article. When an attempt to obtain a lock is unsuccessful, the requestor waits for a random period selected from a uniform distribution; the algorithm doubles the mean of the distribution after each failed attempt up to a maximum. At the start of a fresh synchronization period the initial mean corresponds to half of the mean used in the previous period. The maximum mean is set to 16K cycles, which is roughly the time required to service a simple write miss (i.e., three network round trips or approximately 600 cycles) times the number of nodes in the system. I initialize the mean to one cycle, which corresponds also to the minimum mean.

### 3.5.2 Test&test&set

Test&test&set is an extension of test&set that performs a read of the lock before attempting a test&set operation [RS84]. This primitive enables waiting requesters to spin on shared, read-only copies of the lock (local spinning), waiting for the holder to release the lock. When the lock holder issues the release, the read-only copies are invalidated, and having obtained a writable copy of the lock, the holder releases it. Then, all the requesters issue a request to load a read-only

copy of the lock, and finding it released, all attempt a test&set. However, only one will succeed.

Although test&test&set employs local spinning to reduce interconnect traffic while the lock is held, the time needed to acquire the lock is longer than test&set (see Table 3.3), due to the requesters' initial requests for read-only copies (instead of an exclusive copy, as with test&set). The contention when the lock is freed can be substantial, as all requesters attempt to acquire the lock at that point, and then all attempt to upgrade the lock to a writable state. Exponential backoff may therefore improve test&test&set as well as test&set. Collocation with test&test&set may work better than with test&set, since the lock holder can still read data allocated in the lock's cache line, as it is shared with the requesters. Test&test&set collocation is not ideal, however, since the holder will ping-pong the cache line with requesters whenever it writes to the collocated data.

### 3.5.3 MCS locks

The MCS scheme [MCS91a, MCS91b] inserts requesters for a held lock into a software queue at the time of the request, using atomic operations such as swap and compare&swap to update the list correctly. With queue-based locking, arbitration for the eventual recipient of the lock is therefore performed in advance, first-come, first-serve. Arbitration for test&set and test&test&set, conversely, occurs at the time of lock release, increasing the synchronization period.

The price of maintaining the requester queue in software is larger overhead, especially under contentionless conditions. When a lock is released, however,

communication occurs only between the releaser and the requester at the head of the queue. Network traffic is thus reduced to a constant number of network traversals per synchronization access. In addition, each processor waiting for the lock spins locally on distinct memory addresses (instead of a single address as with test&test&set), which further reduces the load on the network. Each processor in the queue maintains a pointer to the address on which the next processor in the queue spins. When the current lock holder leaves the critical section, it simply clears the value pointed to by the address that it maintains.

Since each requester is spinning on a different address, these software queue-based algorithms cannot easily benefit from collocation. Partial collocation can be achieved by placing protected data along with the data structure that tracks the queue insertion point (tail pointer). If there is little contention, partial collocation may be effective. A more sophisticated approach could better exploit collocation by placing data either with the insertion pointer when there is no contention, or with the appropriate queue element when contention exists. However, this approach requires copying of data which, done carelessly, may sacrifice their integrity (e.g., in the context of recursive data structures). I do not investigate this approach. These algorithms are also unable to prefetch data without significant changes that greatly add to their complexity.

### 3.5.4 Anderson's lock

Anderson's proposal implements the queue of waiting processors as a circular array [And89, And90]. This scheme uses fetch&add to assign each processor a

unique entry in the array. Aboulenein and his colleagues [AGGW94] show that Anderson's solution performs no better than the MCS solution; therefore I will not discuss Anderson's primitive any further.

### 3.5.5 Graunke and Thakkar's lock

Graunke and Thakkar's primitive is very similar to Anderson's lock. Like Anderson's, this primitive places each requesting processor in a circular array. The main difference that Graunke and Thakkar's scheme uses swap instead of fetch&add to insert a processor in the array.

### 3.5.6 LH and M locks

Magnusson, Landin, and Hagersten propose two software queue-based locking primitives, LH and M [MLH94] (independently, Craig propose also a lock identical to LH [Cra93]). They claim that their primitives will require one fewer remote access to transfer a lock than does MCS, enabling their schemes to outperform MCS when lock contention exists. The LH lock achieves this behavior at the expense of increased latency to acquire an uncontested lock. The M lock achieves the more efficient lock transfer without increased uncontested lock access latency, at the expense of significant additional complexity in the lock algorithm. I implement both locks according to the description in their paper, which presents the algorithms in detail [MLH94].

### 3.5.7 QOLB

Goodman, Vernon, and Woest discuss the Queue-On-Lock-Bit primitive (QOLB—originally called QOSB) [GVW89], which is the first proposal for a distributed, queue-based locking scheme. QOLB maintains a hardware queue of waiting processors, in which pointers to adjacent queue entries are held in the cache line. Waiting processors spin locally on a "shadow" copy of the lock address, preventing unnecessary network traffic or interference with the lock holder. Because lock requesters spin on the same address as that of the lock, without evicting or downgrading the lock holder's copy, effective collocation is possible (unlike the other primitives discussed so far). When the holder releases its lock, the lock is sent directly to the requester at the head of the queue, incurring a total of one network crossing to transfer the lock (see Table 3.3).

In addition to enabling local spinning, collocation, and efficient hand-offs through queueing, QOLB is a non-blocking primitive. Therefore QOLB can issue synchronous lock prefetches, allowing the processor to overlap data and lock access times with other useful work. If the prefetch is issued sufficiently far in advance, it is possible for the requester to see no overhead associated with the

critical section entry, either for accessing the lock or the data. Figure 3.2 shows an example of how QOLB is used to access data in a critical section. The first call to `enqolb` (a non-blocking operation) allocates a shadow copy of the cache line and sends a message that inserts the requester into the hardware requester queue. This early request allows the processor to overlap the fetch time with useful computation. The subsequent calls to `enqolb` in the loop spin locally until the owner releases the lock and sends it directly to the waiting node. When `enqolb` returns "true," the processor enters the critical section. The processor relinquishes the lock with the call to `deqolb`, at which point both the lock and any data in the lock's cache line are sent directly to the next waiting processor. In this example, I assume that the critical section data can fit in 60 bytes. This will not always be the case, of course. Also, QOLB is fair in general, except in the unusual

```
type monitor = (
    lock : int,
    data : char[60]                    [Assumes 64-byte cache lines.]
)

Critical Section(monitor : ref monitor) {
    enqolb(monitor→lock)               [Prefetch lock & data.]
    [Various computation here.]
    ...
    while (¬enqolb(monitor→lock)) {}   [Spin if necessary and acquire.]
    [Critical section here.]
    ...
    deqolb(monitor→lock);              [Release lock.]
}
```

**Figure 3.2 QOLB code example.**

cases when a processor's shadow copy of the lock is replaced from its cache, forcing the processor to rejoin the queue at its end.

### 3.5.8 Lee and Ramachandran lock

Lee and Ramachandran also discuss a hardware queue-based locking primitive [LR90]. Their primitive is very similar to QOLB, however, its design is restrictive: the primitive only works on bus-based systems. Indeed, Lee and Ramachandran's primitive takes advantage of the broadcast nature of the bus to support read locks in addition to exclusive locks. The primitives discussed so far only support exclusive locks, which allow only a single processor at a time in the critical section. Read locks, used often in database and operating systems, relax this restriction by permitting multiple processors in a critical section. Programs use this primitive when a processor needs only read access to the data, but still requests data consistency with respect to potential writers. Since this thesis focuses strictly on exclusive locks, I will not measure the performance of Lee and Ramachandran locks. When used strictly in the context of exclusive locking, these locks will perform in a way similar to QOLB.

### 3.5.9 Fine-grain data prefetching and forwarding

Numerous studies demonstrate the potential benefits of fetching data into the cache ahead of time (data prefetching) [CB94, FP91, GHG+91] and writing data directly into a remote cache (data forwarding) [KCPT95, PY94, RSS+95]. Abdel-Shafi and his colleagues [ASHAA97], and Trancoso and Torrelas [TT96] specifically consider the use of these two types of primitives to speed up the execution of

synchronization operations. The first group of researchers improve the hand-off behavior of MCS under contention using data forwarding. Under contention, the current lock holder can generally identify the processor wanting the lock next. In effect, the lock holder can, upon release, signal directly the next processor. Thus, this technique can potentially reduce to one the number of messages that MCS requires to transfer the lock (contrast this figure with the number in the third column of Table 3.3).

### 3.5.10 Reactive synchronization

Reactive synchronization [LA94, Lim95] dynamically switches among software algorithms that perform well under various levels of contention. For instance, it may combine test&set for low-contention phases of execution with MCS for periods of high-contention. Reactive synchronization attempts to achieve both low latency lock access and efficient transfer at low cost (e.g., using only all-software primitives).

I implement reactive synchronization, closely following the guidelines in the paper [LA94]. For low-contention phases, I use test&set with exponential backoff. For high-contention phases, I use MCS (the results presented later show that MCS is the best-performing software lock under high contention, of the locks that I measure). My implementation switches to MCS after five consecutive lock acquisitions experienced higher levels of contention than a fixed threshold (a mean delay of 32 clock cycles). I switch from MCS to the low-contention lock when the queue is empty upon lock release five consecutive times.

## 3.6 Experimental evaluation

I measure the performance of the six synchronization primitives discussed in Section 3.5, varying mechanisms from Section 3.2 when possible, except that I do not simulate collocation in conjunction with the LH and M locks (it will become obvious later in this chapter that MCS generally performs just as well as LH and M, which are not inherently more amenable to collocation than MCS). I also measure the performance of reactive synchronization (also without collocation since there is no straight forward method to apply this optimization with this primitive). The seven locking schemes (and their corresponding abbreviations) are as follows: test&set (denoted TS), test&set&set (denoted TTS), MCS locks, LH locks, M locks, reactive synchronization (denoted R), and QOLB. I use the following abbreviations for optional mechanisms or policies: collocation (+C), hand-inserted synchronous prefetch (+P), and exponential backoff (+E).

### 3.6.1 Methodology

**Target system.** The simulation platform for this study is the Wisconsin Wind Tunnel (WWT) described in Chapter 2. The target systems that I simulate are cache-coherent shared-memory systems. For the experiments involving the microbenchmark, the size of the simulated system ranges from 1 to 64; for all the other experiments the system always comprises 32 nodes. I typically run each set of experiments twice; once assuming a sequentially consistent memory system and a second time assuming an aggressive implementation of release consistency that attempts to remove inefficiencies associated with writes completely.

**Microbenchmark.** I measure the raw critical section execution time using the first microbenchmark described in Chapter 2. The benchmark accesses the critical section a total of 3,200 times. Once in the critical section, a processor waits 800 cycles before releasing the lock (this stall simulates access to, and computation of, protected data). After release, the releasing processor waits for a random amount of time selected from a uniform distribution.[1] The mean of the distribution is five times the critical section delay (4,000 cycles) and the minimum is zero cycle. As the number of nodes increases, the contention for the lock increases, and eventually the reduction in execution time stops (and in some cases reverses) owing to the increase in lock contention. For this experiment, I assume a fixed network latency between any two nodes of 100 cycles and a sequentially consistent memory system. Relaxing the memory ordering constraints may benefit synchronization algorithms that require multiple memory accesses to acquire or release the lock. Algorithms in this category include most notably the software queue-based locks such as MCS. However, I do not simulate the execution of this microbenchmark with a more aggressive memory model. Indeed, the goal of this experiment is not to compare the precise execution time of the different locking alternatives quantitatively, but rather to compare their performance in relative terms and study their trends and behavior under varying contention levels. Macrobenchmarks simulations include experiments with more aggressive memory models and these experiments show that relative performance does not change substantially.

1. The selected times are pseudo-random and repeatable for all experiments.

**Macrobenchmarks.** The benchmark applications that I use for these experiments are Barnes, Mp3d, Ocean, Pthor, Raytrace and Water-Nsq. Descriptions of these benchmarks appear in Chapter 2. I list the problems that the benchmarks solve and the inputs that I use in Table 3.4.

For these macrobenchmarks, I vary the memory model as well as the synchronization primitive. By using two memory models (sequential consistency and release consistency), I show that the performance gained by improving the synchronization primitive cannot also be gained solely by making the memory model more aggressive. The memory models that I simulate are two different implementations of release consistency: sequential consistency (denoted SEQ), and an aggressive implementation that attempts to minimize the number of times that the processor stalls for memory write operations (denoted REL). For the latter memory model, I label all memory accesses as aggressively as possible according to the structure proposed by Gharachorloo and his colleagues [GAG+92, GLL+90], and insert the appropriate memory fences to achieve release consistency on the simulated hardware platform. Although the system assumes blocking loads, I implement a merging write buffer of up to 64 non-blocking stores, which allows multiple stores to be coalesced and loads to be serviced by stores. This large buffer permits very aggressive relaxation of the consistency model for stores. Specifically, this buffer allows load and store memory operations to bypass earlier incomplete stores and allows loads to read the buffer content early (i.e., before the corresponding pending writes complete). This reordering specification is similar

**Table 3.4 Macrobenchmarks.**

| BENCHMARK | INPUT |
|---|---|
| Barnes | 2,048 bodies, 11 iter. |
| Mp3d | 24,000 mols, 25 iter. |
| Ocean | 98×98, 2 days |
| Pthor | `risc`, 1,000 timesteps |
| Raytrace | `teapot` |
| Water-Nsq | 512 mols, 3 iter. |

to the Partial Store Ordering (PSO) relaxed memory consistency model [SFC91, CS99].

I simulate a merging buffer large enough (64 entries) such that results will not be affected by its size. Undoubtedly, for these applications and for the system simulated, this buffer is overdesigned: my simulation results indicate that on average 1 to 4 entries are used (however, there are points in the benchmarks' executions where more than 60 entries are consumed).

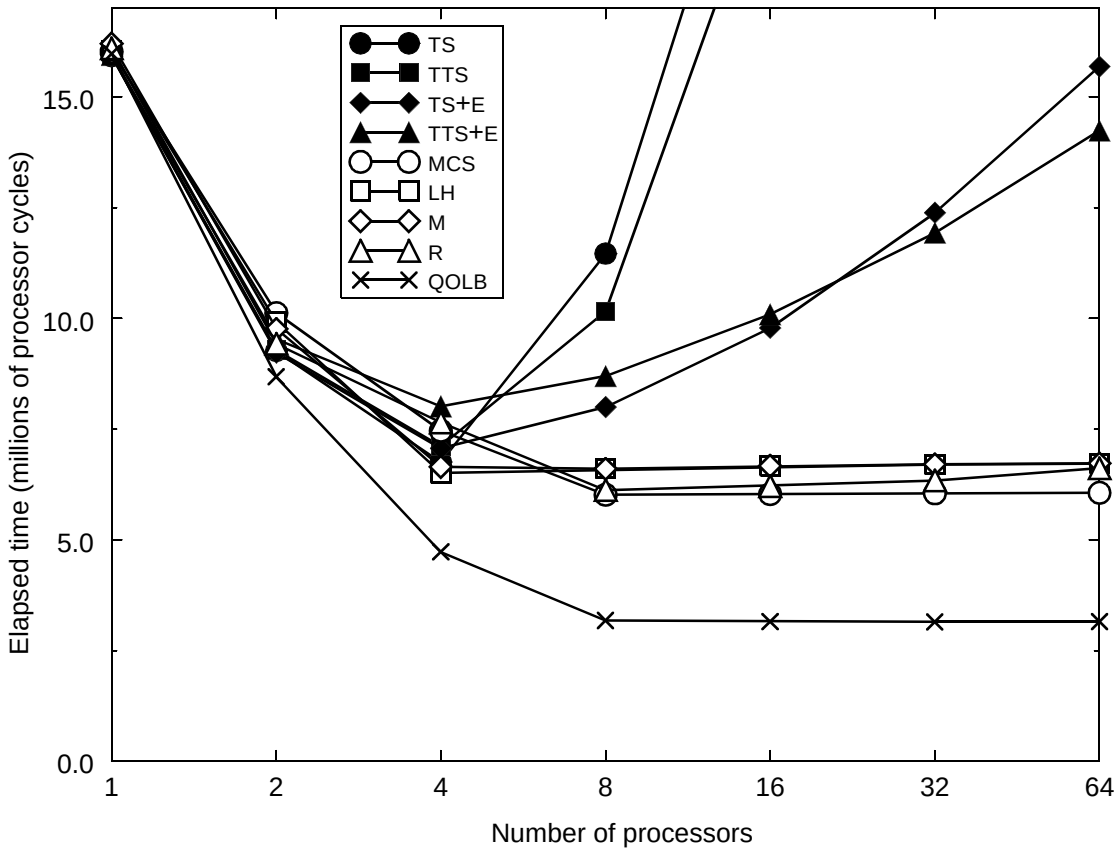
### 3.6.2 Microbenchmark results

Completion time of the microbenchmark loop is plotted in Figure 3.3. Since there is no shared data used in the critical section of this benchmark, I do not explore the benefit of the collocation mechanism here. I measure the completion time of test&set and test&test&set both with and without exponential backoff, MCS, LH and M locks, QOLB, and reactive synchronization (using test&set with exponential back-off for the low-contention case and MCS for the high-contention case). We see that QOLB performs best in all cases, under both low and high con-

tention. Test&set and test&test&set perform well under low contention (one or two processors), but their performance quickly degenerates for more than four processors. Excluding QOLB, all other primitives perform comparably under medium contention (up to 4 processors).

The general trends for the performance of all the queue-based synchronization primitives (MCS, LH, M, QOLB) display the behavior that we have come to expect: gradual performance improvement until the sequential executions of the critical section dominate the performance of the microbenchmark. Close inspection reveals that LH and M do not consistently outperform MCS under high contention, contradicting the claims of their authors. Indeed, Magnuson, Landin and Hagersten [MLH94] state that under high contention, MCS generates one extra cache miss than do LH or M. However, careful collocation of the MCS "next" pointer and the lock bit, as implied by the original article [MCS91a], can prevent this extra cache miss. Under high contention, this collocation permits two read accesses to be satisfied by a single read miss.

Under high contention, MCS slightly outperforms both LH and M. The difference in performance is attributable to the cache behavior of these primitives and the cache coherence protocol I simulate. Under MCS, a processor always reuses the same queue element (or memory address) to insert itself in the queue. Under both LH and M, queue elements tend to migrate from releasing to acquiring nodes [MLH94]. In SCI, a write to a migrating cache block requires more network transactions than does a write to a block accessed mostly by one processor (see last column of Table 3.3). Other cache-coherence protocols may affect the performance of




**igure 3.3 Microbenchmark performance.**

these primitives differently. For instance, in a DASH-like protocol, performing a write to a block shared by *N* other processors always takes the same amount of messages independent of which processor added itself to the sharing list last. Therefore, in a DASH-like system there would not be any difference between the performance of MCS and that of the M or LH locks.

Finally, we see that this implementation of reactive synchronization scheme is successful in that it closely tracks the performance of the best software alternative under both low- and high-contention conditions.

### 3.6.3 Macrobenchmark results

I present the results of the macrobenchmark experiments in Table 3.5. Test&set is the base case for each benchmark and memory model. I list the simulated execution time of each base experiment (in millions of processor cycles, discounting initialization) in parentheses in the test&set (TS) row and sequentially consistent memory system (SEQ) column of Table 3.5. In the last row of this table, I also list the absolute speedups (with respect to the uniprocessor run) for the QOLB experiment with all optimizations enabled and assuming a sequentially consistent memory. Note that Pthor scales particularly poorly even with the use of an efficient synchronization primitive (QOLB). The nature of the logic circuit being simulated is such that Pthor often processes very few events (less than 10) before entering a deadlock, the resolution of which requires a global synchronization through a barrier. The relatively high frequency of global synchronization precludes satisfactory scaling behavior for Pthor, at least for the chosen input. The other numbers in Table 3.5 are all speedups relative to their particular base case.

What is most striking about these results is the fact that test&test&set with exponential back-off and collocation enabled performs nearly as well as QOLB in all benchmarks except for Mp3d. Indeed, for these benchmarks, QOLB never outperforms test&test&set by more than 4% (sequentially consistent Raytrace) and for one benchmark (Pthor with relaxed consistency) test&test&set is 9% faster than QOLB. Yet, these two primitives achieve their respective performance with different means. QOLB achieves its performance with very efficient lock hand-off and deferred transfer of the lock; while test&test&set achieves its performance

**Table 3.5 Speedups of synchronization primitives.** The numbers in parentheses represent the execution time (in millions of clock cycles, discounting initialization) for the particular benchmark running on sequentially consistent hardware. The numbers in brackets represent the absolute speedups for the QOLB experiment with all optimizations enabled. The other numbers represent relative speedups, calculated as the ratio of the execution time (discounting initialization) of the base run to that of the optimized synchronization primitive.

| | BENCHMARK | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Barnes | | Mp3d | | Ocean | | Pthor | | Raytrace | | Water-Nsq | |
| EXPERIMENT | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL |
| TS | (175) | 0.94 | (146) | 0.99 | (16.3) | 1.19 | (123) | 1.12 | (548) | 1.36 | (65.2) | 1.02 |
| TS+C | 1.76 | 1.98 | 1.05 | 1.15 | 1.31 | 1.70 | 0.76 | 1.04 | 9.61 | 10.62 | 1.04 | 1.06 |
| TS+E | 1.55 | 1.58 | 1.41 | 1.71 | 1.12 | 1.38 | 1.15 | 1.33 | 4.49 | 4.88 | 1.02 | 1.04 |
| TS+E+C | 1.84 | 1.98 | 1.60 | 1.98 | 1.31 | 1.70 | 1.29 | 1.59 | 12.29 | 13.77 | 1.01 | 1.04 |
| TTS | 1.02 | 1.11 | 1.09 | 1.13 | 1.02 | 1.22 | 1.08 | 1.23 | 1.05 | 1.18 | 1.00 | 1.02 |
| TTS+C | 1.83 | 1.99 | 1.15 | 1.25 | 1.33 | 1.71 | 0.90 | 1.46 | 11.46 | 11.99 | 1.04 | 1.06 |
| TTS+E | 1.55 | 1.59 | 1.30 | 1.62 | 1.11 | 1.40 | 1.12 | 1.31 | 4.30 | 4.90 | 1.02 | 1.04 |
| TTS+E+C | 1.86 | 1.97 | 1.48 | 1.86 | 1.27 | 1.71 | 1.32 | 1.63 | 12.53 | 13.50 | 1.02 | 1.04 |
| MCS | 1.71 | 1.73 | 1.44 | 1.60 | 1.24 | 1.56 | 1.20 | 1.32 | 7.02 | 6.15 | 1.02 | 1.04 |
| MCS+C | 1.73 | 1.76 | 1.61 | 1.76 | 1.26 | 1.67 | 1.41 | 1.57 | 6.66 | 6.91 | 1.02 | 1.03 |
| LH | 1.69 | 1.72 | 1.30 | 1.51 | 1.25 | 1.56 | 1.17 | 1.36 | 6.36 | 6.45 | 1.00 | 1.02 |
| M | 1.70 | 1.71 | 1.09 | 1.36 | 1.25 | 1.56 | 1.17 | 1.27 | 6.34 | 6.45 | 1.01 | 1.03 |
| R | 1.70 | 1.75 | 1.16 | 1.48 | 1.19 | 1.50 | 1.19 | 1.36 | 6.61 | 7.10 | 1.01 | 1.03 |
| QOLB | 1.92 | 1.95 | 2.01 | 2.27 | 1.31 | 1.67 | 1.26 | 1.49 | 13.03 | 13.31 | 1.04 | 1.05 |
| QOLB+C | 2.04 | 2.07 | 2.63 | 2.81 | 1.34 | 1.72 | 1.56 | 1.89 | 14.95 | 15.41 | 1.05 | 1.06 |
| QOLB+C+P | 2.04 | 2.06 | 2.64 | 2.79 | 1.32 | 1.69 | 1.59 | 1.90 | 14.90 | 15.50 | 1.06 | 1.07 |
| | [22.9] | | [7.16] | | [16.7] | | [1.64] | | [20.7] | | [27.1] | |

with very few instructions combined with collocation to optimize data transfer and exponential back-off to reduce the impact of lock contention. Note, however, that, in a production-quality application, collocation should probably not be used along with a primitive like test&test&set. First of all, performance improvement

due to collocation cannot be guaranteed with a primitive that does not implement synchronous lock transfer; moreover, in cases of heavy contention, performance degradation could be dramatic. The purpose of studying collocation with test&test&set in this thesis is not to prove such an alternative viable, but rather to investigate the potential of collocation to reduce synchronization inefficiencies.

Also striking about these results is the range of speedups, considering that the only two parameters being varied are the synchronization primitive and the assumed memory model. Three benchmarks record speedups greater than 100% (Barnes, Mp3d, and Raytrace). The largest and smallest speedups measured, with all four mechanisms employed, are sequentially consistent Water-Nsq and Raytrace respectively: QOLB with collocation and prefetch enabled speeds up the execution of Water-Nsq by 6% and Raytrace by a factor of 15.5.

The large speedups obtained by certain primitives are not necessarily as much an indication of their worth as evidence of the large inefficiencies that a poor synchronization primitive can introduce. For instance, all but one primitive improve the performance of Raytrace by at least a factor of 4. Queueing effects on Raytrace's main lock structure and SCI's deficient support for widely-shared data combine to lead to test&set and test&test&set's substantially inferior performance compared to the other primitives. Raytrace has one lock that it accesses much more often than any other locks in the program. Raytrace uses that lock to assign each new ray that it spawns a unique identifier using a global counter. The high number of rays required to render a scene with the relatively low number of instructions executed per ray makes this critical section a highly contested

resource. Therefore, the use of a naive synchronization primitive can easily lead to processors spending an inordinate amount of time queued waiting for the lock. For instance, without collocation, the critical section execution time augments substantially. Indeed, since there is contention for the critical section, the local cache is unlikely to have a copy of the global counter just as a processor obtains the lock. Therefore, the fetch of a copy of the counter will delay the execution of the critical section. Since the critical section has only a few instructions (load the counter into a register, increment it, and store the new value back to memory) the fetch delay will dominate the execution time of the critical section. The longer the critical section the more likely a synchronization request will find the lock busy. On the other hand, collocation decreases the execution time by about two orders of magnitude, reducing considerably the number of times that a request finds the lock busy. In the case of test&set, the traffic generated by the processors competing for the lock further degrades performance. While local spinning helps test&test&set somewhat with this extra traffic, the release of the lock is substantially slowed down by the fact that SCI must invalidate the sharers one at a time [Kax98]. Other coherence protocols typically use broadcast to invalidate sharers, which would improve the release performance of test&test&set, and improve the performance of test&test&set over test&set somewhat.

Water-Nsq is relatively insensitive to any of the applied optimizations. Water-Nsq is a compute-intensive benchmark that communicates little. Such programs do not need optimized synchronization because there is so little communication involved. The speedups for Ocean are small not because the mechanisms are inef-

fective, but because Ocean uses locks less frequently than the other benchmarks do (see Table 3.4).

For all benchmarks, QOLB with collocation consistently captures the bulk of the performance improvement to be gained. The implementation of synchronous prefetching has no impact on speedup. There may be more opportunity for improvement with synchronous prefetch, if the codes or algorithms are restructured to exploit the power of the QOLB prefetch operator.

Nearly all benchmarks exhibit similar performance for QOLB and test&set (or test&test&set) with collocation; this is untrue for Mp3d and Pthor, however. Using collocation with test&test&set improves the performance of Mp3d little, and deteriorates the performance of Pthor. The lower performance of Pthor with collocation results from the relatively long length of Pthor's critical sections. These long critical sections give requesters the opportunity to attempt obtaining the lock, pulling both the lock and critical section data out of the holder's cache. This behavior does not occur with QOLB since QOLB defers the lock (and collocated data) transfer until the current lock holder leaves the critical section.

Partial collocation with MCS improves the performance of all benchmarks, except for Barnes and the sequentially consistent runs of Ocean and Raytrace. In these cases collocation either has little impact (Ocean and Barnes) or degrades performance slightly (Raytrace). Unlike test&set, MCS causes only a fixed number of memory operations to be issued per synchronization access, thus limiting the disturbance caused by collocation.

Raytrace exhibits much larger speedups than does any other benchmark. The Raytrace base case (test&set) is extremely slow (as is test&test&set). Adding any other mechanism besides local spinning improves the performance of Raytrace substantially. These two primitives perform so poorly because much of the locking is for very small critical sections, for which there is heavy contention. Collocation makes the small critical sections extremely fast. Queue-based locking eliminates the large relative overhead that occurs due to contention when the lock is released.

Adding exponential backoff improves performance moderately for all benchmarks but Mp3d and Pthor in the sequentially consistent runs, in which I observe slowdowns of up to 20%.

Reactive synchronization is generally within 25% of the best performing synchronization primitives (disregarding the collocation mechanism and the QOLB runs). The exceptions are the sequentially consistent runs of Barnes and Mp3d, where reactive synchronization is 32% and 53% slower than MCS, respectively.

### 3.6.4 Individual mechanisms

This section isolates the performance contributions of the individual mechanisms in Section 3.4. Figure 3.1 shows performance differences between eight pairs of experiments (for each benchmark). Each pair of experiments isolates one particular mechanism. There is doubtless interaction between an “isolated” mechanism and the other components of the synchronization primitive. This decomposition is not intended to quantify the performance contribution of indi-

vidual mechanisms definitively, but to aid in understanding of how each of them affect performance. I also isolate the exponential backoff policy. I list the isolated mechanisms or policies in Table 3.6, along with their corresponding experiment pairs.

**Table 3.6 Experiment pairs.**

| | EXPERIMENT PAIR | |
|---|---|---|
| ISOLATED MECHANISM OR POLICY | With | Without |
| Local spinning | TTS | TS |
| Exponential backoff | TTS+E | TTS |
| Queue-based locking | MCS | TTS |
| | QOLB | TTS |
| Collocation | TS+C | TS |
| | TTS+C | TTS |
| | QOLB+C | QOLB |
| Synchronous prefetch | QOLB+C+P | QOLB+C |

All runs in Figure 3.4 assume a sequentially consistent memory model. The y-axis plots speedup. Figure 3.4 shows that local spinning is generally ineffective. Queue-based locking (using MCS) increases speedup for all benchmarks. Using collocation with test&set and test&test&set causes very different behavior across the benchmarks: reducing speedup (Pthor), having a negligible effect (Mp3d), causing a moderate increase (Ocean), and causing a large increase (Barnes and Raytrace). This high variance with collocation exists because requesters may either steal the data from the lock holder, hurting performance, or prevent extra

remote transfers into a network filled with arbitration traffic, thus mitigating exceptionally poor performance.

Synchronization prefetching is ineffective, never affecting the running time by more than 2%. As mentioned before, there is more opportunity for improvement with synchronous prefetch, as I do not restructure the codes or algorithms to exploit the power of the QOLB prefetch operator.

## 3.7 Future technology

In a previous study Mellor-Crummey and Scott conclude [MCS91b] that "special-purpose synchronization mechanisms such as the [QOLB] instruction are

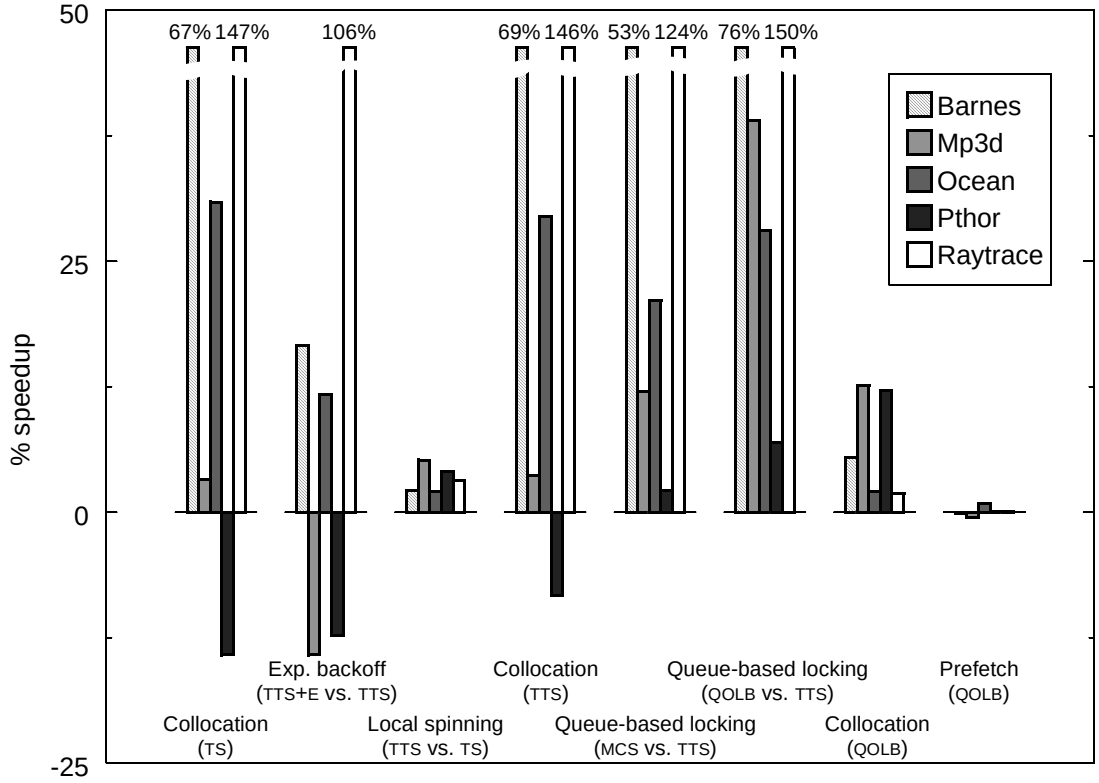



**Figure 3.4 Effect on individual locking mechanisms.**

unlikely to outperform our MCS lock by more than 30%." This claim does not agree with the results in Section 3.6, where QOLB improves the performance by 40%. Furthermore, this claim becomes more suspect as processors become faster, system size increases, and the relative delays of interconnect traffic increases. This is borne out by the results for future technology assumptions, where improvements gained by using QOLB for synchronization are greater than 100%. This performance improvement is likely to increase as processors grow even faster, suggesting that special synchronization hardware support such as QOLB will be more important in future shared-memory multiprocessors. I note, however, that high-performance processor designs that use aggressive prefetching and speculative execution may well be able to capture some of the same latency reduction achieved by QOLB.

Water-Nsq was relatively insensitive to any of the synchronization primitives, since it is a compute-intensive benchmark that communicates little. Because there is so little communication involved, Water-Nsq shows the greatest improvement in performance when run with future technology assumptions, which speeds up the base run by a factor of 4.44. Speeding up the processor relative to the network latencies, however, not only increases the execution speed dramatically; it also increases the benefits of optimization. With the future technology assumptions, the combined speedup from QOLB with collocation, prefetch, and a sequentially consistent memory system increases from 6% to 13%. The critical observation is that even computation-bound jobs eventually become communication-bound, either as improvements in the processing speed outstrip gains in the

network or as these programs are scaled to larger systems. Thus, the optimizations investigated here eventually become effective even for such programs.

**Table 3.7 Speedups of synchronization primitives assuming future technology parameters.** The numbers in parentheses represent the execution time (in millions of clock cycles) for the particular benchmark running on sequentially consistent hardware. For each primitive, this table shows its speedup versus TS, calculated as the ratio of the execution time of the base run to that of the optimized synchronization primitive. The last row shows the speedup of the future technology, seq/TS simulation to that for the current technology with SEQ,TS

| | BENCHMARK | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Barnes | | Mp3d | | Ocean | | Pthor | | Raytrace | | Water-Nsq | |
| EXPERIMENT | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL |
| TS | (281) | 0.96 | (303) | 1.11 | (26.6) | 1.22 | (227) | 1.13 | (1130) | 1.62 | (73.4) | 1.05 |
| TTS | 1.03 | 1.27 | 1.18 | 1.19 | 1.03 | 1.26 | 1.08 | 1.25 | 1.11 | 1.24 | 1.03 | 1.06 |
| MCS | 2.04 | 2.07 | 1.62 | 1.76 | 1.31 | 1.72 | 1.23 | 1.35 | 7.68 | 7.78 | 1.06 | 1.09 |
| QOLB | 2.41 | 2.46 | 2.32 | 2.63 | 1.40 | 1.89 | 1.29 | 1.52 | 15.00 | 15.30 | 1.09 | 1.11 |
| QOLB+C | 2.64 | 2.70 | 3.22 | 3.36 | 1.44 | 1.95 | 1.61 | 1.99 | 20.41 | 21.92 | 1.11 | 1.13 |
| QOLB+C+P | 2.64 | 2.69 | 3.23 | 3.38 | 1.41 | 1.92 | 1.65 | 1.97 | 20.15 | 22.59 | 1.13 | 1.16 |
| TS: cur/fut | 3.12 | | 2.40 | | 3.06 | | 2.70 | | 2.43 | | 4.44 | |

## 3.8 Small caches

Results discussed so far used a 1MB cache, which is a reasonable size for current technology. Assuming a large cache, benchmarks with small data sets may overemphasize the benefits of improvement in the locking efficiency. Thus, this section discusses results for runs assuming current technology and smaller cache sizes that are more reasonable given the input data sets the benchmarks use.

Table 3.8 summarizes these results. The selected cache size fits the second largest working set of each benchmark (usually the most important; refer to Chapter 2 for a discussion of the different working set sizes). The last row of Table 3.8 lists the chosen cache sizes. Two conflicting behaviors affect the performance improvement due to collocation. Placing lock and data in the same cache line increases the effectiveness of the cache when compared to data organization imposed by synchronization primitives that do not lend themselves well to collocation (such as MCS). This observation is countered by the opportunities for QOLB becoming less frequent, because of the longer execution—due to more cache misses—lessening the potential for performance improvement.

I observe both trends in these experiments. The performance improvements attributable to QOLB with collocation for Barnes (factor of 2.2 speedup) is larger than those with the larger cache (factor of 2.0). On the other hand the performance improvements attributable to QOLB with collocation for Raytrace (factor of 5.6 speedup) is substantially lower (factor of 14.6 speedup). Part of the reason for this behavior is that test&set runs better in the smaller cache (speedup of 10%). I believe that the additional misses induced by the smaller cache force lock requests to be more spread apart, which in turn lowers the lock contention improving the performance of test&set. The performance improvements from using QOLB for the other benchmarks are similar to those with the larger cache.

Relaxing memory ordering shows the same general trends as in the 1MB cache experiments, but to a smaller degree. The only exceptions occur with Raytrace and Water-Nsq. For Raytrace, the performance improvements due to QOLB are

**Table 3.8 Speedups of synchronization primitives assuming caches that fit the second largest working set of each application.** The numbers in parentheses represent the execution time (in millions of clock cycles) for the particular benchmark running on sequentially consistent hardware. For each primitive, this table shows its speedup versus TS, calculated as the ratio of the execution time of the base run to that of the optimized synchronization primitive. The last row shows the speedup of the future technology, SEQ/TS simulation to that for the current technology with SEQ,TS

| | BENCHMARK | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Barnes | | Mp3d | | Ocean | | Pthor | | Raytrace | | Water-Nsq | |
| EXPERIMENT | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL | SEQ | REL |
| TS | (188) | 0.94 | (146) | 0.99 | (22.8) | 1.10 | (143) | 1.09 | (499) | 1.31 | (80.1) | 1.02 |
| TTS | 1.02 | 1.09 | 1.10 | 1.15 | 1.02 | 1.12 | 1.06 | 1.15 | 1.09 | 1.27 | 1.01 | 1.02 |
| MCS | 1.81 | 1.83 | 1.29 | 1.60 | 1.17 | 1.31 | 1.14 | -.-- | 4.92 | 4.96 | 1.02 | 1.04 |
| QOLB | 1.81 | 1.83 | 2.02 | 2.26 | 1.21 | 1.35 | 1.22 | 1.37 | 5.32 | 5.30 | 1.04 | 1.05 |
| QOLB+C | 2.16 | 2.19 | 2.57 | 2.76 | 1.22 | 1.37 | 1.33 | 1.50 | 5.60 | 5.35 | 1.04 | 1.06 |
| QOLB+C+P | 2.16 | 2.18 | 2.58 | 2.77 | 1.18 | 1.34 | 1.33 | 1.50 | 5.66 | 5.46 | 1.00 | 1.03 |
| CACHE SIZE | 64KB | | 4KB | | 2KB | | 2KB | | 2KB | | 2KB | |

lower with the more relaxed memory model. This result is perhaps evidence for the fact QOLB might be able to capture a lot of the benefits of relaxing memory ordering constraints in applications that use locks frequently. For Water-Nsq the performance improvements using a sequentially consistent memory system are comparable for both cache sizes, primarily because the benefits are so small (<7% speedups).

In general, the conclusions drawn earlier in this chapter remain valid for the smaller cache size.

## 3.9 Summary

This chapter focused on providing efficient locking primitives to improve the performance and scalability of fine-grain shared-memory parallel programs. Instead of focusing on the individual latencies associated with mutually exclusive accesses to critical sections, I focused on the global execution time of critical section accesses. I defined the notion of a synchronization period: one "cycle" of multiple serialized accesses to a critical section. I broke this time into three phases (Transfer, Load/compute, and Release), and classified the components of each of these phases as either unavoidable latencies or removable inefficiencies. I identified four optimizing mechanisms (local spinning, queue-based locking, collocation, and synchronous prefetch) that can assist in eliminating the removable overheads of critical section accesses.

I performed a thorough evaluation of this space, simulating the performance of sixteen locking constructs (formed from six base primitives: test&set, test&test&set, MCS, LH, M, and QOLB) in detail with both real parallel applications and the more traditional microbenchmarks.

The results showed that local spinning consistently aids performance but not very much. Queue-based locking was very effective, except in the cases where the overhead of MCS, LH, and M locks hurt low-contention critical section access latencies. Collocation of the lock and locked data in the same cache line showed wildly different effects for test&set and test&test&set; collocation may greatly increase

or decrease performance, depending on the benchmark. Synchronous prefetching was the least effective of any of the mechanisms.

Perhaps the most important result of these experiments is the large improvements due to collocation applied in combination with appropriate synchronization primitives. Collocation consistently improved the performance of QOLB, and test&set and test&test&set, both with exponential back-off. In fact, test&test&set with exponential back-off and collocation performs nearly as well as QOLB without collocation for all benchmarks except Mp3d. While applying the collocation with test&set or test&test&set in production-quality applications may not be necessarily wise; it could be applied successfully to high-performance carefully-tuned applications.

Another important result of these experiments is the consistent and large performance gain that QOLB achieves, which is further increased by collocation and by future technologies. Graunke and Thakkar [GT90] concluded that "... elaborate hardware [synchronization] schemes are unnecessary even when considering larger non-bus-based [systems]." Mellor-Crummey and Scott stated [MCS91b] that "special purpose synchronization mechanisms such as the [QOLB] instruction are unlikely to outperform our MCS lock by more than 30%." The results in this chapter contradict these assertions; QOLB outperforms MCS by 40% for Mp3d and more for future technologies.

Lim and Agarwal claimed [LA94] that reactive synchronization "reduces the motivation for providing hardware support for queue locks." Since QOLB outperforms the best software locks under either low- or high-contention conditions, it

should also outperform reactive synchronization schemes. These results confirm this hypothesis—QOLB speedups were from 10% to 92% higher than reactive synchronization, and this disparity only increased by adding collocation and synchronous prefetch to QOLB.

The performance improvements gained by more efficient synchronization primitives are even more dramatic when the system parameters are adjusted to reflect technology trends. Assuming faster processors that are more heavily penalized by network delays, the benefits to be had from better synchronization primitives increase, suggesting that efficient synchronization primitive may become an important feature of future systems.

# Chapter 4

# Implementation of synchronization primitives

## 4.1 Introduction

The success of hardware support for programming languages or operating systems may depend critically on the ability of an application or the operating system to interact effectively and correctly with the proposed hardware support. In particular, scalability of codes that synchronize frequently may depend on the efficacy of a locking primitive implementation. A poor implementation can lead to an imbalance between useful computation and synchronization overhead. This imbalance will thwart further performance improvements that could be had from adding more processing elements or refining the computation grain between synchronization events. Also, an implementation must be correct. For example, a synchronization protocol should not be able to access another program's lock variable, and the implementation must ensure forward progress and avoid deadlocks.

This chapter discusses the correctness and performance issues associated with implementing efficient support for synchronization primitives. Correctness issues

include (1) protection, (2) forward progress, and (3) deadlock freedom. If the synchronization state is shared among users of a multiprogrammed system, this system must provide methods to enforce protection among untrustworthy programs. Possible solutions can depend on existing protection support or rely on a special-purpose design or use a combination of both. In a multiprocessor, locking support implies a distributed solution among the nodes in the system. Nodes in a multiprocessor cannot necessarily observe events instantly as they occur and do not necessarily observe these events in the same order. This property, common among all distributed systems, raises forward progress and deadlock concerns. When appropriate, I discuss these concerns and review possible solutions. Once a decision is reached about the set of locking mechanisms desired on a system, the design team has a choice of implementation strategies for these mechanisms; each of these strategies has a different cost and each can reach a certain performance level. For each mechanisms, I review several implementation alternatives and discuss their cost/performance trade-offs.

The rest of this chapter is organized as follows. Section 4.2 discusses the *hardware* mechanisms required to implement primitives incorporating all four synchronization mechanisms introduced in Chapter 3. This section also reviews alternatives for implementing these mechanisms and discusses their cost/performance trade-offs. Section 4.3 puts it all together, and describes and evaluates an all-software implementation of QOLB on a cluster of unmodified workstations. Section 4.4 reviews related work and Section 4.5 summarizes.

## 4.2 Hardware mechanisms for synchronization

I identify six hardware mechanisms that a multiprocessor system must implement to support an efficient synchronization primitive that incorporates all the four locking mechanisms introduced in Chapter 3. These hardware mechanisms are: (1) *naming*, (2) *protocol processing*, (3) *synchronous cache-to-cache transfer*, (4) *placeholder ("shadow line") allocation*, (5) *non-blocking instructions*, and (6) *association of a lock and the data it protects*. In the next subsections, I give more detailed descriptions of these mechanisms and review implementations alternatives. Table 4.1 summarizes the six identified hardware mechanisms along with

**Table 4.1 Summary of implementation alternatives for each hardware mechanism.**

| COST | HARDWARE MECHANISM | | | | | | PERFORMANCE |
|---|---|---|---|---|---|---|---|
| | naming | protocol processing | synchronous cache-to-cache transfer | placeholder allocation | non-blocking instructions | lock & data association | |
| lower → | conventional memory | all-software | asynchronous | main memory | system call | spatial locality | ← lower |
| | | | | | call into software library | | |
| | special synchronization memory | | | | | variable cache line size | |
| | | programmable protocol processor | synchronous | off-chip cache | dual mapping | | |
| ← higher | | | | | new ISA | mapping | higher → |
| | | hard-wired full custom | | on-chip cache | | | |

the reviewed implementation alternatives. I associate with each alternative a relative cost (entries nearer the bottom of the table are relatively more costly). Note, however, that this cost assignment is subjective and that other persons may come up with orderings different than the one presented in this table. For instance, some may be of the opinion that additions to a current instruction set architecture (ISA) is not very expensive, while some others may argue that such additions have hidden costs (such as the cost of having to carry these additions in all future implementations of the new instruction set).

### 4.2.1 Naming

Typically, applications access more than one synchronization variable. Therefore, systems traditionally support a synchronization name space, in which a program can allocate many synchronization variables. Synchronization schemes may choose to store locks in conventional memory or use a special synchronization memory. Prevalent designs allocate locks in conventional memory. Then, these locks can be accessed by normal load and store instructions as well as by any other (possibly atomic) memory instructions that operate on memory (e.g., test&set or swap). This approach has the following advantages. First, the name space is virtually inexhaustible (i.e., there are as many possible locks as there are memory locations). Second, no special provision is necessary to save or restore the synchronization state of a process; this state is stored as part of physical memory, of which the operating system is already aware. Third, protection is ensured by the mechanisms already in place (e.g., virtual memory support). Finally, in this

context, there is an obvious method to achieve collocation (i.e., place lock and data in the same cache line). However, storing locks in conventional memory has at least one disadvantage: this approach may complicate the address computation of elements in certain types of data structures. For instance, if a programmer associates a lock per row of an array and stores each lock next to its corresponding row, then the address computation of each array element is going to be more complicated than in the case of a regular array.

Another alternative to naming synchronization variables is to allocate them in a special memory and provide means to operate on them. This special memory may take different forms: (1) it may be a distinct name space accessed with special instructions (e.g., Sequent's SLIC gates [BKT87]) or (2) it may be a tag associated with each conventional memory location accessed with conventional load and store instructions with atomic side effects (e.g., HEP full/empty bits [Smi81]). The advantage of this alternative is that locks and data structures may be composed arbitrarily without introducing undue complications in data structure address computation. However, this approach suffers from several disadvantages. First, it may restrict the number of locking variables. If a program require more locks than provided, it must somehow schedule this limited resource. A possible scheduling policy is to create virtual locks, the access to which is protected by a subset of the hardware locks. Second, the state of a process is now more elaborate and must include the locks allocated in the special memory. Third, if this special memory is shared among users, protection mechanisms must be provided. If the tagged memory approach is used, protection may be provided through the sup-

port available for conventional memory (e.g., virtual memory). On the other hand if the distinct memory approach is used, specific support may be required. Finally, the association of a lock and the data it protects to support the collocation locking mechanism may be more complex to achieve in the case of a distinct synchronization name space.

### 4.2.2 Protocol processing

For any synchronization primitive implementation, the processor and the memory hierarchy must interact properly and efficiently to achieve both correctness and a satisfactory performance level. Typically, such interaction is specified using a protocol. A protocol is a specification that consists of a list of states and events, a catalog of possible actions, and a set of permissible transitions. The latter set indicates, for a given event, into what states the protocol can move and what actions need taking place for each transition. A combination of sound design methodology, theorem proving, and simulation and verification tools are used to devise a protocol that is both correct (e.g., deadlock free) and that performs well. If the components used to build a multiprocessor do not support a given synchronization operation, the system designer must provide the corresponding locking protocol by different means.

There are two primary alternatives to performing protocol processing: either a hard-wired state machine or a programmable protocol processing engine can perform the task. Note that there are many possibilities for hybrid designs.

The first alternative represents the traditional approach: almost all bus-based systems implement the cache coherence protocol this way. The trade-offs of hard-wired implementations are well known: potential for high performance and manufacturing cost in exchange for a certain inflexibility. Many manufacturers market processors that include an on-chip memory hierarchy and an implementation of a coherence protocol to maintain the consistency of data stored on-chip. For such implementations, there is an obvious path to add efficient support for synchronization to their products: extend the existing cache coherence protocol to include support for locking. Such a design is perfectly feasible as demonstrated by the IEEE SCI standard [IEE93]. SCI defines a basic directory-based distributed coherence protocol; SCI also defines optional extensions to the basic protocol to provide a range of systems with varied cost and performance. One such option is the support for the QOLB synchronization primitive.

The other alternative—protocol processing using a programmable engine—is recently strongly advocated by Heinrich and his colleagues [HKO+94], and by Reinhardt, Larus, and Wood [RLW94]. This approach trades some performance loss against the flexibility to change or extend the protocol at any time. Reinhardt, Larus, and Wood go even further and expose the necessary mechanisms to allow programmers or compilers to tailor the behavior of shared memory to the need of applications. Indeed, the user can supply custom protocols that can maintain cache coherence differently for distinct regions of memory. In his thesis, Reinhardt discusses implementation alternatives of this flexible scheme that range from low cost (but low performance) to high performance (at a higher price)

[Rei96]. At one extreme, he describes an all-software system without any special hardware support beyond the capability to send and receive messages. In this environment, the (compute) processor itself performs the protocol processing. At the other extreme, he proposes a tightly integrated system composed of one or more (compute) processors, a network interface, and a dedicated custom-designed protocol processor.

### 4.2.3 Synchronous cache-to-cache transfer

To implement queue-based locking very efficiently, a system must be able to transfer data directly from a node's local memory to another's. In addition, to support synchronous transfer in general and synchronous prefetch in particular, this data transfer must occur at a moment of the current data holder's choosing. This ability to defer the data transfer introduces supplementary forward progress and deadlock issues in addition to those already present in conventional data movements: the transfer may occur at an indeterminate time after the initial request (including never, in case of a programming error). Such unbounded delays on data transfers must not impede the forward progress of other operations in the system nor should they lead to a deadlock. In particular, resources at the different stages of request and data transfer need to be carefully allocated and deallocated. One solution to prevent deadlock and ensure forward progress is to decouple the request from the data transfer completely. In this scheme, buffer resources at the different network interfaces are only allocated for the duration of the request transfer. A new set of resources is allocated and released for the dura-

tion of the data transfer. Consequently, no network resources remain reserved for more than the duration of a network round-trip (i.e., a network transfer and its acknowledgement).

From an implementation point of view, the two important aspects of synchronous cache-to-cache transfer are (1) point-to-point message transfer and (2) deferred transfer. The latter aspect is under the control of the cache coherence protocol. If synchronous transfer is desired, the protocol must be designed to support it. Unfortunately, if the protocol is hard-wired and tightly integrated (e.g., on a processor chip) synchronous transfer may not be easily achievable. The second aspect (point-to-point decoupled message transfer) must be supported by the underlying interconnecting network. The two primary means of connecting nodes in a multiprocessor system are buses and point-to-point networks. Point-to-point messages are trivially supported by point-to-point networks, however these networks are typically more expensive and reserved for large system configurations. In contrast buses are much less expensive because of a simpler design and large volumes of production that foster economy of scale. Unfortunately, not all buses support decoupled operations. For instance, many early buses (e.g., the DEC PDP-11 Unibus, the IBM PC-AT bus, or Intel's Multibus) do not support decoupled requests/responses (also called split transactions). However, with the advent of aggressively pipelined memory systems, more and more buses support decoupled operations necessary for the implementation of the synchronous cache-to-cache transfer mechanism.

### 4.2.4 Placeholder allocation

To support both local spinning and synchronous data transfers (used both for synchronous prefetch and efficient data transfer in implementations of the queue-based locking mechanism) a system must provide the capability to allocate a placeholder in its local memory for the requested data. This allocation serves a purpose that is very similar to the allocation of a cache line upon a cache miss. Mostly, this mechanism avoids deadlocks and promotes forward progress. It does so by allocating buffer space ahead of time to ensure that the message containing the requested data does not block upon arrival waiting for a location to store its content. Also, in the context of synchronization, placeholder allocation plays a unique role. First, it allows the node to spin locally without generating requests. Finally, it serves as a notification device to inform the waiting node that the lock has arrived.

The placeholder can be allocated either in a small local memory near the processor (e.g., a cache) or in main memory (a portion of which can be managed as a large fully-associative cache). The mechanisms for allocating space in main memory are inexpensive and exist already through conventional memory storage allocators.

The other alternative to allocating a placeholder in main memory is to reserve storage in a local memory. This solution may be somewhat more expensive to implement but affords superior performance. A small local memory near the processor has an access time that is faster than main memory. Ideally, the placeholder should be allocated as close to the processor as possible (i.e., on-chip).

However, this implementation may be too costly (it implies a custom designed on-chip memory hierarchy that supports efficient synchronization primitives). Instead, a less costly implementation may provide locking support through an off-chip local memory, in which case placeholder allocation need only be provided for that local memory.

### 4.2.5 Non-blocking instructions

An application must be able to issue a synchronization operation that appropriate parts of the hardware platform must recognize as such. In addition, to support synchronous prefetch, these instructions must not block.

In the absence of proper synchronization instructions in the instruction set architecture (ISA) it is possible to simulate the behavior of locking primitives in the operating system through the system call interface. The advantage of this approach is that it also provides protection for free. Unfortunately, the system call interface is typically very slow and is unlikely to perform adequately for fine-grain synchronization operations.

An alternative proposal, described in detail in the works of Blumrich and his colleagues [BDFL96], and Markatos and Katevenis [MK97], but known long before these works appeared, consists of creating dual virtual mappings for those pages in shared-memory that contain synchronization variables. The seed of this idea comes from the desire to use regular load and store instructions to command an external device. Performing this task correctly is challenging. For instance, an access may remain externally invisible in the presence of an on-chip cache. Simi-

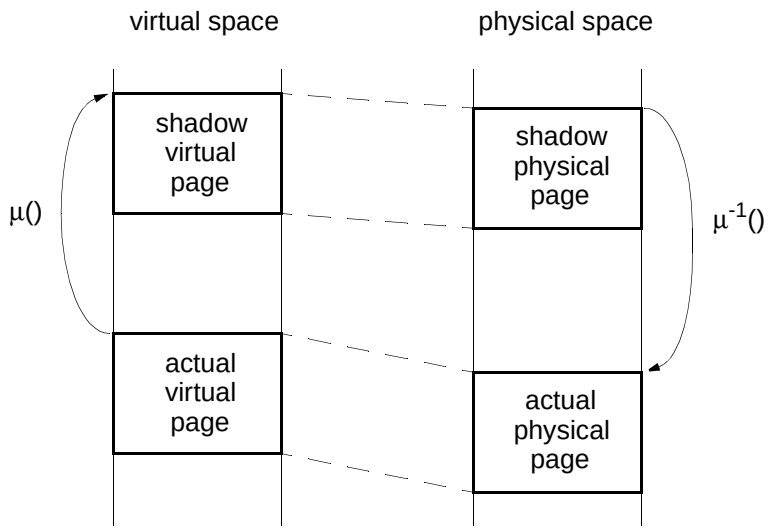



**Figure 4.1 Memory and shadow mappings.**

larly, writing a virtual address on the data bus is of little use if the external device does not know how to translate a virtual address.

The basic idea of dual mappings is to create two translations for each address that may refer to a synchronization variable. The first mapping is the conventional mapping tying a virtual page to a physical page. The operating system maintains a second mapping so that an external observer can distinguish synchronization from conventional memory operations (reading from and writing to memory). Figure 4.1 illustrates the concept. To any virtual page in which an application may issue a synchronization operation corresponds a shadow virtual page. The function μ() maps each actual virtual page to a dual shadow virtual page. A virtual page and its shadow counterpart map to distinct pages in the physical space. The operating system assigns backing storage to the actual pages

but not to the shadow pages, since the latter do not contain any data. While the content of the actual physical page may be cached, the content of the shadow physical page remains uncached. Therefore any access performed in the shadow space will always be visible externally. An external observer can recognize accesses performed in the shadow space and, given a judicious choice for the function μ(), can translate any shadow physical address back to the corresponding actual physical address.

The function μ() must be invertible and $\mu^{-1}()$, the inverse of μ(), should be relatively simple to compute to allow for cheap hardware implementation. A possible function μ() might employ an unused upper bits of the address space as follows (assuming addresses no larger than $2^{32}$):

$$\mu(addr) = 1 \parallel addr_{0\ldots30}$$

the inverse of μ() is simply

$$\mu^{-1}(addr) = addr_{0\ldots30}$$

Given this methodology, QOLB-like instructions could be defined as follows:

$$\texttt{enqolb}(addr) \equiv \texttt{load}(\mu(addr))$$

$$\texttt{deqolb}(addr) \equiv \texttt{store}(\mu(addr))$$

This scheme has serious deficiencies. In particular, it bypasses the entire on-chip memory hierarchy to communicate with an external device. Therefore, the on-chip memory hierarchy is not used efficiently for these operations: the requested data will be stored in the off-chip memory hierarchy upon arrival and

only when the processor actually accesses the lock will the data move on-chip. Also, in the worst case, it may double the number of existing mappings and double the translation bandwidth, which may burden the virtual memory system in an unacceptable way.

A third alternative to providing synchronization instructions is to extend an existing instruction set. This solution provides the best performance alternative; however, the associated cost can be substantial and the addition of new instructions may slow down other operations in the microprocessor. Additional cost include extended decode hardware, more complex control logic, and added testing requirements.

### 4.2.6 Association of a lock and data

To benefit from collocation, the application must have some means to inform the system about the relationship between a lock and the data it protects.

Many of the synchronization primitives discussed so far (e.g., QOLB) achieve collocation by taking advantage of the “prefetching” capability of cache lines larger than a word. Indeed, by carefully placing lock and data in the same cache line, whenever the memory system sends away the lock, it also ships some data with it. While inexpensive, the limitation of this scheme lies in the fact that all the data associated with a lock may not fit in a single cache line. Therefore only a portion of the data will benefit from the collocation mechanism.

Variable cache line size and binding are two strategies that can boost the effectiveness of collocation. If the size of a cache line is programmable, an application

can select the size that satisfies best the application's needs. Other reasons have already motivated the exploration of variable cache line size [DL92, JMH97]. Chapter 5 explores this solution further.

Another alternative to boost the effectiveness of collocation is to provide mechanisms so that an application can inform the memory system of the bond between a lock and its associated data explicitly. Then lock and data do not need to be stored near one another anymore. Instead the memory system can store the binding information in a table that the system consults each time the application issues a synchronization operation.

## 4.3 Putting it all together

I decompose overall design alternatives of efficient synchronization support into four classes representing a varying range of cost and performance. These classes are: (1) all-software systems, (2) systems based mostly on stock components, (3) systems with key components (most notably the microprocessor) redesigned to include special support for efficient synchronization, and (4) fully integrated, aggressive systems achieving a performance ideal close to the shared-memory multiprocessor that I simulate in Chapter 3. For each of these alternatives, I discuss the approach to providing efficient support for synchronization. Then I discuss different options for implementing the six hardware mechanisms described in the previous section. Finally, in the next section I put it all together and show, for each of the four design styles, a possible implementation based on the mechanism implementation options discussed below.

The first design class provides the shared-memory abstraction completely in software, relying only on the underlying hardware's ability to send and receive messages. Systems in that class include Blizzard-S, Blizzard-E, and Blizzard-ES [SFL+94]; CRL [JKW95]; Midway [BZS93]; SAM [SL94a]; Shared Regions [SGZ93]; and Shasta [SGT96]. All-software designs allow for rapid prototyping and may provide testbeds useful for demonstrating a concept (e.g., testing the feasibility of an interface). Later in this section, I will discuss how an all-software solution can implement the four synchronization mechanisms, albeit not very efficiently.

Some academic research groups argue for building computing systems of the second class, that is, based on as many stock components as possible. Faster time-to-market, improved reliability and testing, increasing reliance on third-party software, software and hardware compatibility considerations, and lower manufacturing cost lead manufacturers to rely more on stock components rather than designing components themselves. This view holds particularly for the use of stock microprocessors to build parallel systems. Most parallel systems built today rely on commercial microprocessors. Systems in this class includes university prototypes, for instance Blizzard-T0 [SFL+94] and DASH [LLJ+92, LLG+92] and commercial systems, for instance Sequent's Sting [LC96] and HAL's S1 System [WGH+97]. In the context of stock component-based systems, building hardware optimizations can be challenging. To minimize cost, optimizations have to be implemented either in software within the constraints of an existing and immutable design or in hardware at the periphery (off-chip) of the different off-the-shelf components. Also, these optimizations must not interfere with the perfor-

mance of other activities taking place in the system. Later in this section I will show how to meet the challenges of a low-cost design while still attempting to provide some reasonable performance.

Many processors are available today ready to be integrated in a multiprocessor system. These processors include at least one level of on-chip cache and a complete built-in external interface that allows a system designer to connect a few of these processors together without additional circuitry. These processors maintain consistency of data stored on-chip either by supporting write-through caches [Smi82] or by implementing write-back caches with bus- or directory-based coherence protocol. Two examples of multiprocessor-ready microprocessors are the Intel Pentium Pro [Int96] and the Alpha 21364 [Gwe98]. These processors represent the latest step in a trend towards a continuous on-chip migration of system and multiprocessing functionality. Later in this section I discuss what aspects of these microprocessors need changing to support efficient synchronization.

The last type of system considered corresponds closely to the system that I simulate in Chapter 3. It is an ideal systems in terms of achieved performance of synchronization operations. This system serves as a gauge against which to compare the performance of alternative implementations.

Table 4.1 summarizes the implementations alternatives surveyed in the previous section. I have placed the cheaper solutions towards the top of the table; the ones that are expected to perform better are towards the bottom of the table. The actual positions of the solutions in the table are only indicative; comparisons

across columns are misleading at best. Also, alternative implementations to two different hardware mechanisms are not necessarily compatible.

Table 4.2 shows three overall solutions to the problem of supporting an efficient synchronization primitive incorporating all the four synchronization mechanisms described in Chapter 3. All these solutions assume that the locks are allocated in conventional memory, assume that the network fabric supports synchronous transfer of locks and assume an implementation of collocation that employs the "prefetching" capability of a cache line.

**Table 4.2 Some implementation alternatives.** All solutions in this table assume a network fabric that supports fully decoupled request response messages and assume an implementation of collocation that employs the "prefetching" capability of a cache line.

| | HARDWARE MECHANISM | | |
|---|---|---|---|
| DESIGN ALTERNATIVES | protocol processing | placeholder allocation | non-blocking instructions |
| all software | software | main memory | call in S/W |
| stock components-based | off-the-shelf processors | off-chip cache | dual mappings |
| existing processor design derivative | extension of existing cache-coherence protocol | on-chip cache | new ISA |

The all-software scheme is the least expensive implementation. This solution processes protocol events in software linked with the application; the application invokes synchronization operations by directly calling the appropriate function in the protocol. This solution uses existing virtual memory support to provide necessary protection and address translation. It allocates placeholders in main mem-

ory, a part of which is managed as a large fully-associative cache. Section 4.3.1 describes such an implementation in more detail and evaluates its performance.

A solution based mostly on stock components may use an off-the-shelf processor to process protocol events. The “compute” processor can communicate with the protocol processor with the dual mapping technique. Since this solution bypasses on-chip memory hierarchy, it allocates placeholder in an off-chip cache.

The third solution is based on an existing processor design that already integrates on-chip caches and a cache-coherence protocol to maintain the consistency of on-chip data. Such a processor may already contain most of the mechanisms needed to support an efficient synchronization primitive like QOLB. In addition to extending the existing coherence protocol, this solution requires instructions to trigger the appropriate protocol actions.

### 4.3.1 Proof of concept

To prove that efficient synchronization support can be implemented on today’s hardware I have implemented a synchronization primitive incorporating all four synchronization mechanisms described in Chapter 3 according to the “all software” design described in the first row of Table 4.2. I call this design SOFTQOLB since it is essentially implementing QOLB-like functionality in software. SOFTQOLB is an extension of the Blizzard run-time system developed by Schoinas and his colleagues [SFL+94]. Blizzard uses an executable editing tool [LS95] to support the shared-memory model in software on a distributed-memory multiprocessor. The executable editing tool annotates each access to shared variables to provide

system-wide shared memory. Annotations check that each memory access can complete safely (i.e., data is valid for read accesses and data is writable for write accesses); should a check fail the annotations invoke the appropriate protocol handler to bring the faulting memory location to a suitable state.

To determine if low-cost implementations of QOLB will still outperform other primitives, I compared the performance of five base synchronization primitives. They are test&set, test&test&set, MCS, a message-based centralized queue lock (CQL), and SOFTQOLB, all implemented on a cluster of commodity workstations.

The workstations use the Blizzard run-time system [SFL+94] to provide the illusion of shared memory. Blizzard is an implementation of the Tempest interface [RLW94]. MCS and CQL are part of the locally available Blizzard distribution and are implemented directly on top of the Tempest interface.

I implement SOFTQOLB as an extension of Blizzard's default cache coherence protocol (Stache [RLW94]) by adding support for QOLB. When the program does not invoke synchronization action on a cache line, the protocol defaults to Stache. When the program invokes synchronization actions (through two functions: `enqolb()` and `deqolb()`) the protocol assumes QOLB-specific state transitions. A more detailed description of Stache and SOFTQOLB appear in Appendix A.

I evaluate the performance of the five synchronization primitives (test&set, test&test&set, MCS, CQL, and SOFTQOLB) with the microbenchmark described in Section 2.1.2. The benchmark accesses the critical section a total of 100,000 times evenly distributed among the nodes. Once in the critical section, a processor writes a value into a shared variable, which takes about 80 µs when the variable

is not present in the cache [Sch97]. (This write miss latency assumes 128-byte cache lines.) After the release of the lock, a processor waits for a random amount of time selected from a uniform distribution, the mean of which is approximately 1,400μs (or on the order of 20 times the latency of a write miss). With this benchmark I can evaluate the impact of collocation by modifying the addresses of the lock and the variable in memory. I run the benchmark with all the synchronization primitives without collocating lock and data. Then I rerun test&set, test&test&set, and SOFTQOLB; this time with lock and variable collocated. I measure the time it takes for all processors to complete the microbenchmark iterations. I report the best of three measurements in an attempt to reduce the impact of external perturbation that I have no control over (such as the underlying operating system scheduling other processes). Reinhardt [Rei96] and Schoinas [Sch97] use the same methodology for similar experiments.

I plot the completion time (in seconds) of the microbenchmark loop in Figure 4.2 for a number of nodes ranging from 1 to 16.[1] I measure the throughput of test&set (denoted TS), test&test&set (denoted TTS), MCS, CQL, SOFTQOLB; for test&set, test&test&set, and SOFTQOLB I measure also the throughput with lock and data collocated (denoted with the suffix +C).

When there is no contention, test&set, test&test&set, and MCS perform better than either CQL or SOFTQOLB. The difference is in large part due to the fact that both CQL and SOFTQOLB require invocation of protocol handlers to acquire or

1. Due to a limitation in the Myrinet interface, I could not collect numbers with more than 16 nodes.

release a lock, while the other primitives can perform the same operations using simple SPARC instructions that hit in the cache. Under light contention (two nodes), the performance of test&set and test&test&set is still very competitive, but as the contention increases their performance becomes unstable and degenerates with contention levels beyond five nodes. Collocation sometimes helps test&set and test&test&set, but it is not stable for these primitives as demonstrated by the sudden loss of performance for contention levels of eight and nine nodes. The reason for this performance loss is the sudden increase in the number of cache line invalidations occurring at eight and nine nodes. For these experiments, a node frequently acquires the lock but to relinquish the corresponding cache line immediately to satisfy an invalidation request. After each release, one node is guaranteed to be able to acquire the lock, but the timing of the invalidation requests is often such that these requests are effected before the node has a chance to update the counter collocated with the lock. This coincidence of events increases critical section execution time and the level of contention at the lock leading to an overall increase of the microbenchmark running time.

The queue-based locking primitives perform as expected: with increased contention their performance improves until saturation. Under high contention, SOFTQOLB performs best, MCS poorest.

Under high contention, CQL and SOFTQOLB perform similarly, with CQL being about 10% faster than SOFTQOLB (without collocation). Considering message counts only we could conclude that SOFTQOLB should clearly outperform CQL. Indeed, under high contention, SOFTQOLB transfers the lock in a single hop, while

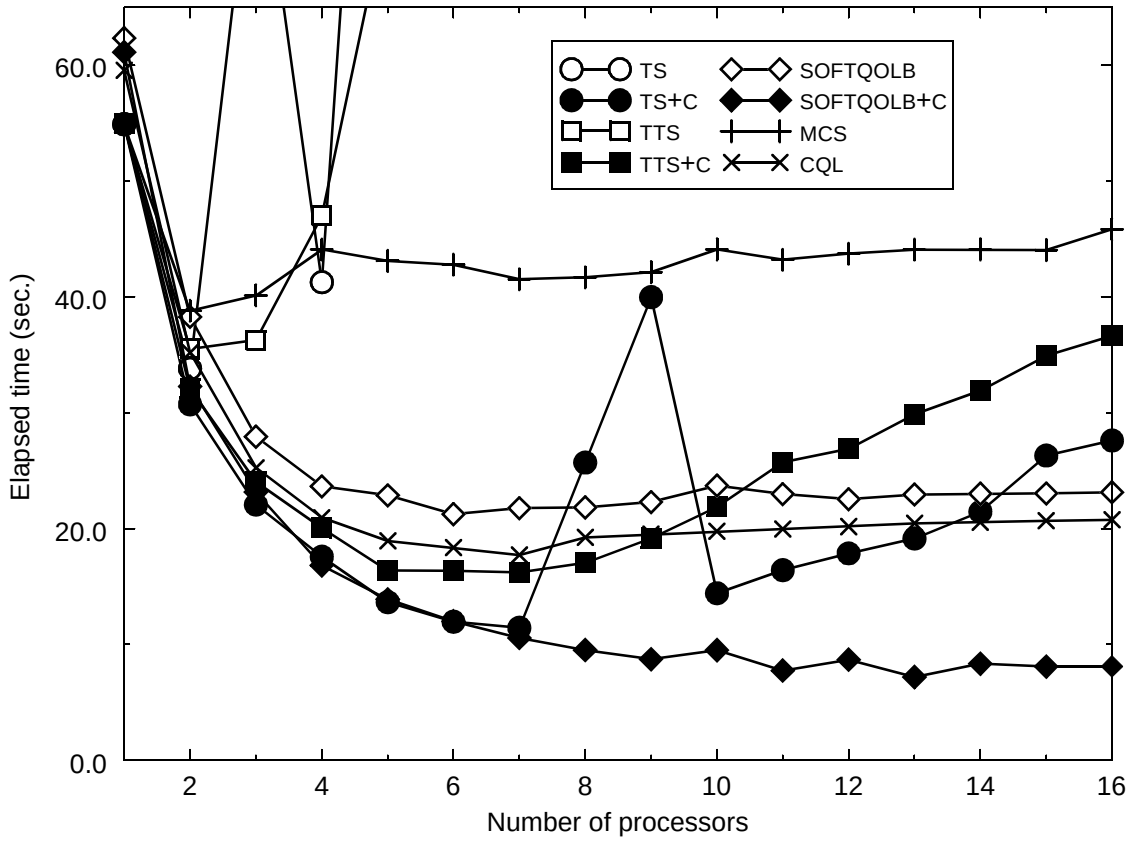



**Figure 4.2 Microbenchmark performance.**

CQL must transfer the lock in two messages (one message from the releaser to the central queue manager and another one from the manager to the acquirer). However, considering message counts only is misleading in this environment because the message transmission times are substantially different. While CQL uses short messages (a single word, four bytes) to transfer the lock, SOFTQOLB transfers entire cache lines (128 bytes). On Blizzard, the short message latency is about half the latency of a message carrying 128 bytes of data (round trip times of ~39µs versus ~80µs [Sch97]).

I also evaluate the performance of MCS, CQL, and SOFTQOLB with the following four applications: SPLASH-2's Barnes, Ocean, Raytrace, and Water-Nsq. A description of these benchmarks and the collocation's strategy appears in Chapter 2. I run these benchmarks with the same problem sizes as in Chapter 3, with the exception of Ocean which I run with a 258×258 grid. I measure the running time of each application discounting initialization; and I report the best of three measurements.

Figure 4.3 plots the running time of the four application benchmarks relative to a sequential execution measured without the overhead of the Blizzard runtime system. I vary the number of processors from one to eight; assuming perfect conditions the relative performance should follow the ideal curve: unit running time with one processor and the running time halved with each doubling of processors.

As already observed with the microbenchmark, in the absence of contention (one node runs), MCS performs better than CQL or SOFTQOLB. MCS can acquire and release a lock with instructions that hit in the cache, while CQL and SOFTQOLB must communicate with the protocol processor to perform the same functions. With increased contention CQL and SOFTQOLB typically outperform MCS, with the exception of Ocean (all system sizes), and Barnes (with two processors). For small system sizes, Ocean generates a lot of traffic [WOT+95] increasing substantially the hardware cache miss rate. The lower cache utilization not only affects the performance of Ocean directly, it also affects the performance of the coherence protocol since it is less likely to find its working set in the hardware cache. SOFTQOLB further aggravates this problem because it is a much larger piece of soft-

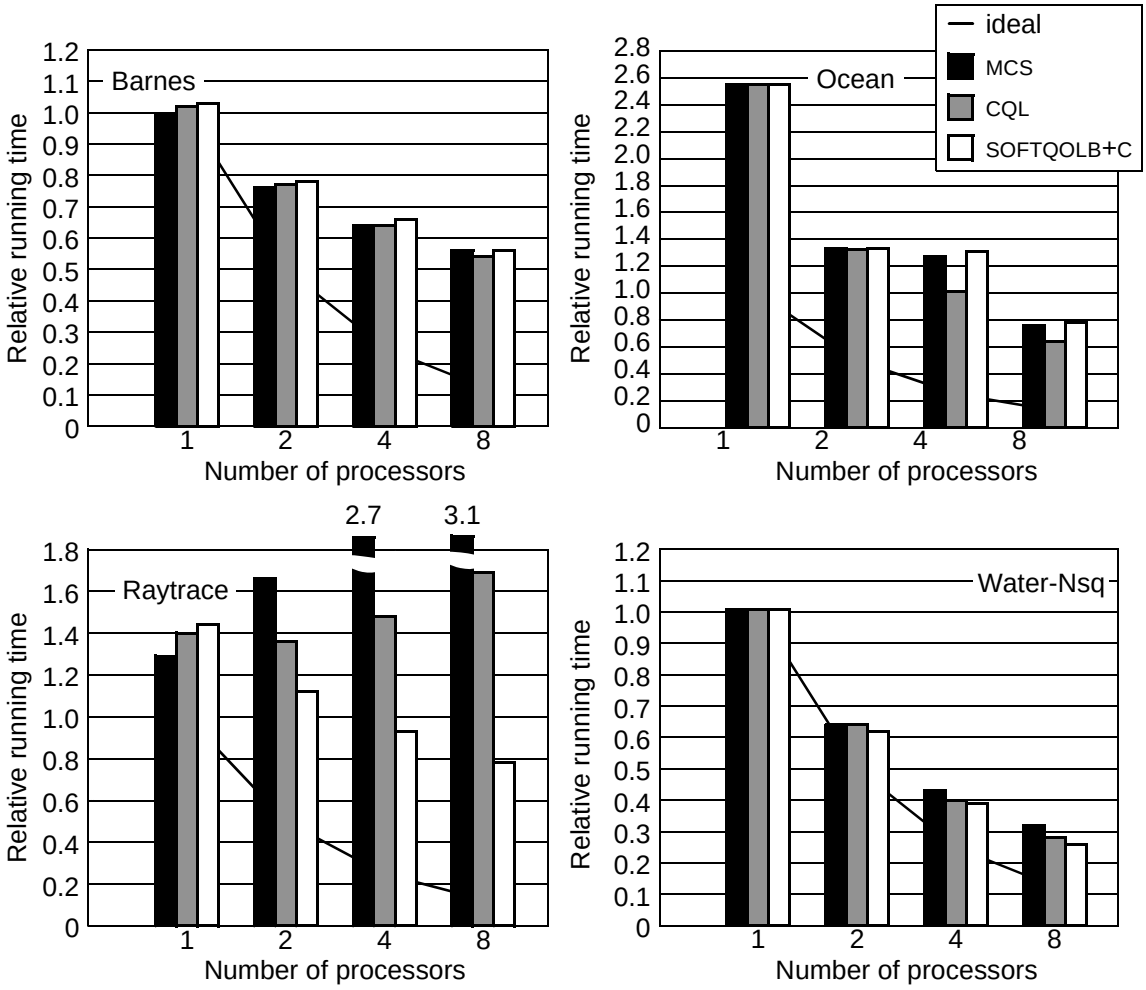



**Figure 4.3 Applications performance.**

ware than Stache. When Barnes runs on two processors the contention to the locks is extremely low. Thus the chances that the same processor accesses the same lock without an intervening access is high (high temporal locality of lock access). For MCS that property translates into a processor likely to find the lock it wants still in its cache; therefore the processor can complete the execution of a

critical section without having to send messages or to communicate with its protocol processor.

In my measurements, SOFTQOLB displays a wide range of different behaviors. In one case (Ocean) SOFTQOLB is markedly worse than CQL (25% slowdown), in other cases, SOFTQOLB is either comparable to CQL (Barnes, 4% slower than CQL) or it is faster than CQL (Raytrace and Water-Nsq, 117% and 8% speedups respectively). The speedup achieved by SOFTQOLB over CQL for Raytrace is very impressive; however, the absolute speedup with respect to the sequential execution is still only 25% for eight nodes.

## 4.4 Related work

The idea of dual mappings to communicate with external devices is an old idea that was doubtlessly "rediscovered" several times. To my knowledge, the first use of dual mappings appears in the Stanford DASH [LLG+92], which uses them to provide a mean to distinguish between regular reads and reads-with-intent-to-modify. Other implementations of dual mappings appear also in the Thinking Machines' CM-5 [TMC91] to send commands to the vector units, in the AP1000 multicomputer to initiate data transfer, in the Stanford FLASH multiprocessor [HGDG94] to initiate user-level DMA transfers, and in the Wisconsin's Typhoon-0 prototype [RPW96] to modify the fine-grain access control bits.

The first careful description and study is due to the members of the Princeton SHRIMP project [BDFL96, BLA+94]. They use these ideas to support very efficient user-level DMA transfers. The SHRIMP system uses user-level DMA transfers to let

applications send and receive messages with minimal operating system intervention. Markatos and Katevenis extend the Princeton work by removing the need to modify the operating system [MK97].

Bitar and Despain first proposed a special cache coherence protocol to support efficient synchronization [BD86]. Their bus-based protocol supports efficient busy-wait (i.e., local spinning), collocation, and synchronous prefetch. Their protocol also supports queue-based locking. However, instead of storing the queue information in the caches, the protocol just records the fact that one or more lock requests have been seen. Then upon lock release, the waiting processors arbitrate to decide which one acquires the lock next. This technique will work as well as the queue-based locking implementations reviewed so far as long as two conditions hold: (1) arbitration is fair and (2) arbitration can be performed out of the critical path.

## 4.5 Summary

This section focused on the implementation challenges and alternatives to supporting efficient synchronization on current and future hardware platforms. I discussed in detail the six hardware mechanisms (naming, protocol processing, synchronous cache-to-cache transfer, placeholder allocation, non-blocking instructions, and lock and data association) that a multiprocessor must implement in order to support an efficient synchronization primitive that incorporates all the four synchronization mechanisms introduced in Chapter 3. For each of these hardware mechanisms I discussed implementation alternatives. Based on these

implementation alternatives, I also described four classes of designs that range over varying cost and performance. These design points are: all-software, stock component-based, derived from an existing processor design, and an ideal system resembling the multiprocessor that I simulated in Chapter 3.

I believe that the inherent cost requirements of a synchronization primitive like QOLB are not prohibitive. Hardware queue-based locking is not prohibitively expensive, as DASH implemented one such synchronization scheme [LLG+92] (it differs from QOLB in that the centralized memory directory keeps track of queued requesters). QOLB is an integral part of the SCI standard [IEE93], and uses many of the same mechanisms needed to implement the coherence protocol. I showed that already today many processors are multiprocessor ready and already implement most of the hardware support needed to implement efficient synchronization primitives (i.e., synchronous cache-to-cache transfer, placeholder allocation, and lock and data association); and that two other mechanisms could easily be introduced (non-blocking instructions) or could be extended from an existing design with little effort (protocol processing).

Finally, I described and evaluated an all-software implementation of QOLB on a cluster of workstations. This design demonstrated the feasibility of providing the functionality of QOLB on unmodified hardware, however because of its inefficiencies this design failed to realize the performance potential of the four synchronization mechanisms introduced in Chapter 3.

# Chapter 5

# A detailed study of collocation

## 5.1 Introduction

Collocation, one of the four synchronization mechanisms introduced in Chapter 3, provides the opportunity to transfer a lock and the data it protects as a single unit. With perfect collocation, when the current owner releases the lock, the releaser sends not only the lock but also *all*[1] the protected data to the next requester, if any. Upon receiving the lock, the new owner can start useful computation immediately, avoiding delays associated with misses when accessing protected data. Indeed accesses to these data will be satisfied locally, since the data arrived with the lock. This scenario approaches the ideal of a minimal synchronization period: the locking throughput becomes completely determined by two fundamental components of synchronization: lock transfer and critical section computation; all the other components, including read and write misses incurred

1. I assume that a critical section accesses most of the data that are protected by the corresponding lock. If a lock protects a large region of memory and only a small subset of that data is accessed each time the lock is acquired, then collocation may not be a good idea.

while executing a critical section are eliminated. Note that most studies do not consider these misses as part of synchronization inefficiencies. This thesis makes a strong case that (1) synchronization inefficiencies should include misses incurred while executing in a critical section and that (2) collocation permits elimination of this source of inefficiency.

In my opinion, collocation has not received yet an adequate treatment in the literature. This study constitutes an initial step towards a better understanding of the advantages and limitations of collocation. I believe that the first persons to recognize the value of collocation are Bitar and Despain with their paper describing a cache coherence protocol that supports synchronization operations explicitly [BD86], and later, Goodman, Vernon, and Woest with their proposal discussing QOLB [GVW89]. Indeed, both proposals embody the observation that lock and data have two distinct access patterns. On one hand, processes access locks asynchronously, that is independently of the status of these locks. On the other hand, a process accesses protected data synchronously, that is only after acquiring the lock successfully. Therefore, unless a synchronization primitive takes into account these two types of access patterns, collocation will not lead to consistent performance improvements.

For instance, it is possible, but not necessarily wise, to collocate lock and data in conjunction with test&set. If there is absolutely no lock contention among processors, test&set will consistently benefit from collocation. However, if there is contention, collocation may degrade the performance of test&set. By repeated but futile applications of test&set to a lock already owned, a process may impede the

progress of the lock owner in the critical section. Indeed, each synchronization request not only steals the lock from the owner's cache, it also takes protected data away, the access of which is unnecessarily delayed.

On the other hand, a primitive like QOLB encourages the collocation of lock and data. Because QOLB takes note of synchronization requests but defer acting upon these requests until the lock is released, collocation will consistently improve synchronization and hopefully overall performance. Specifically, when a process successfully enters a critical section, the corresponding lock will remain available locally until the process releases the lock. Therefore any data collocated with the lock will also remain accessible locally without requiring global communication. If, concurrently, QOLB receives a synchronization request, QOLB enqueues the request for later consideration by notifying the current lock holder. The latter will forward lock and data as soon as it releases the lock.

I believe that collocation is an important optimization that merits careful study. In fact, because collocation helps eliminate unnecessary global communication, collocation is likely to become even more important as the gap between the performance of processors, on one hand, and the performance of memory and interconnection network, on the other hand, is growing.

The organization of the rest of this chapter is as follows. Section 5.2 reviews known collocation strategies and describe their shortcomings. Section 5.3 proposes VAQUM, a new synchronization primitive that incorporates a new collocation strategy that overcomes some these shortcomings. Section 5.4 describes CLEAN, an implementation of VAQUM, and Section 5.5 provides an evaluation of

this implementation. Finally, Section 5.6 reviews related work and Section 5.7 summarizes the contributions of this chapter.

## 5.2 Known collocation strategies

To my knowledge, existing collocation strategies fall into two categories. The first strategy consists of prefetching critical data at the beginning of a critical section execution. The other strategy consists of taking advantage of the "prefetching" capacity of a cache line.

### 5.2.1 Prefetching as collocation

Trancoso and Torrellas propose to insert prefetching instructions at the beginning of a critical section; these instructions send early requests for critical data [TT96]. A compiler can place these prefetching instructions automatically. This strategy works as follows. Processes execute these instructions immediately after successfully obtaining a lock. Hopefully requesting data early helps overlap fetch latencies with the execution of critical section instructions that do not require access to shared data. This strategy also helps with critical data spanning across multiple cache lines since processes send requests for these lines concurrently. Note that a dynamically-scheduled processor with a non-blocking memory hierarchy subsumes this strategy, although care must be taken to assure that data is not fetched before the lock is obtained. Indeed, by being able to issue multiple instructions concurrently, out-of-order processors essentially achieve the prefetching prescribed by Trancoso and Torrellas.

The main drawback of Trancoso and Torrellas' method is that there is not necessarily enough work in the critical section to hide completely the latencies associated with fetching critical data. In fact, delay is almost inevitable, because good programming practice virtually guarantees that an operation that could have been performed without accessing the protected data will not be performed at the beginning of the critical section. Programs could prefetch critical data even before obtaining the lock; however this method would only work if there is seldom contention to the locks.

### 5.2.2 Cache lines as collocation enabler

Goodman, Vernon, and Woest describe QOLB, an efficient synchronization primitive [GVW89]. They suggest implementing QOLB as an extension of a cache coherence protocol. Cache coherence protocols maintain data consistency at a fixed granularity of a cache line that is typically larger than the machine word. To put it another way, any action by a coherence protocol involving data movement does not affect just a single word but an entire cache line. Taking advantage of this fact, cache-coherence-based implementations of QOLB can achieve collocation by judiciously placing data in the same cache line as the lock. In that way, whenever QOLB transfers a lock into the cache of a requesting processor, QOLB also conveniently stores useful data there. Thus, when a program enters the critical section it will find the collocated data locally avoiding global delays associated with fetching the data from a remote location.

The drawback of this strategy is that the benefit of collocation will be sensitive to the size of the cache line used in a particular system. Specifically, if the protected data does not all fit in the same cache line as the lock, the data located outside the line containing the lock will not benefit from collocation. Prefetching can help overcome this limitation. The user or a compiler can identify the parts of the critical data that do not fit in the same cache line as the lock and insert instructions to prefetch these data parts at the beginning of the critical section.

In the rest of this section, I quantify the impact of the cache line size on the performance of collocation. Using WWT, I measure the performance of collocation for the five benchmarks (Mp3d, Ocean, Pthor, Raytrace, and Water-Nsq) described in Chapter 2 using three different choices of line sizes: 32, 64, and 128 bytes.[1] Reusing the same methodology that I applied in Section 3.6.4, I isolate the contribution of collocation to synchronization performance by measuring the speedup of a run with collocation enabled over a run without collocation. Specifically I compare QOLB runs with and without collocation. I measure these speedups for each of the three chosen cache line sizes (32, 64, and 128 bytes).

I recompile each application to eliminate false sharing for a given line size. I do not apply the prefetching scheme described earlier that helps request early data items that cannot benefit from collocation. The simulation models data movement and resource contention accurately up to the point where messages are injected into or retrieved from the network. Using the methodology described in

1. WWT cannot simulate systems with a choice of the cache line size greater than 128 bytes.

Section 2.3 I determine the constant network latency based on runs assuming 64-byte lines (different values for experiments with and without collocation). I then use the same constant value for each simulation with a different cache line size. This assumption is likely to underestimate the performance of runs assuming 32-byte lines and overestimate the performance of runs assuming 128-byte lines. However, the goal of these experiments is not to quantify the impact of line size on the performance of collocation absolutely, but to aid in understanding how their combination affects performance.

For each application, Figure 5.1 shows three pairs of bars, the heights of which are proportional to the running time of each program's parallel section, and are normalized to the run of the corresponding application with 64-byte lines and collocation disabled. From left to right, the three pairs corresponds to the 32-byte, 64-byte, and 128-byte cache line runs, respectively. I measure the speedup afforded by the collocation mechanism and display- that number expressed as a percentage on top of the right bar of each bar pair.

Three applications (Ocean, Pthor, Water-Nsq) show little impact of the choice of line size on the performance of collocation. As noted earlier, Ocean associates little data with each lock (at most a floating point accumulator of eight bytes) which always fit in a cache line along the lock. Moreover I assume the same constant network latency for each of the experiments with a different cache line size. Using different network latencies would likely benefit smaller cache line sizes; however the difference in performance is unlikely to be substantial since Ocean uses locks infrequently. Pthor uses locks for two purposes. One type of lock

guards accesses to global queues; collocation places a lock along with the pointer to the first enqueued element. In this case, the choice of line size has limited impact on the benefit of collocation. Pthor also uses locks to protect data associated with each element being simulated. An element data structure occupies approximately 64 bytes. For this data structure, using 32-byte cache lines should lead to an increase in the execution time of the critical section accessing the circuit elements. However, the length of that critical section is already very long, thus relatively insensitive to additional misses. Water-Nsq associates a lock per molecule and a molecule data structure is about 512 bytes. Accordingly, the performance of collocation improves with larger line sizes; however the improvement is marginal in part due to the fact that Water-Nsq communicates little.

The other benchmarks (Mp3d and Raytrace) appear more sensitive to the selection of the cache line size. In Mp3d, the data associated with each lock will not fit in a single cache line until its size reaches at least 64 bytes; hence the performance of collocation improves markedly relative to 32-byte lines when the cache line size is either 64 or 128 bytes. Raytrace displays a strange behavior: without collocation it performs best with a 64-byte cache line, with collocation it performs best with a 128-byte cache line. The reason is that the dynamic behavior of the different data structures under varying line sizes is bimodal. Raytrace associates little data (one word) with the important lock, therefore that data structure favors smaller line sizes. On the other hand, the scene and ray data structures are larger favoring larger lines. The combination of collocation and a 128-byte

cache line produces the best result by efficiently transferring protected data and exploiting the spatial locality of the scene and ray data structures.

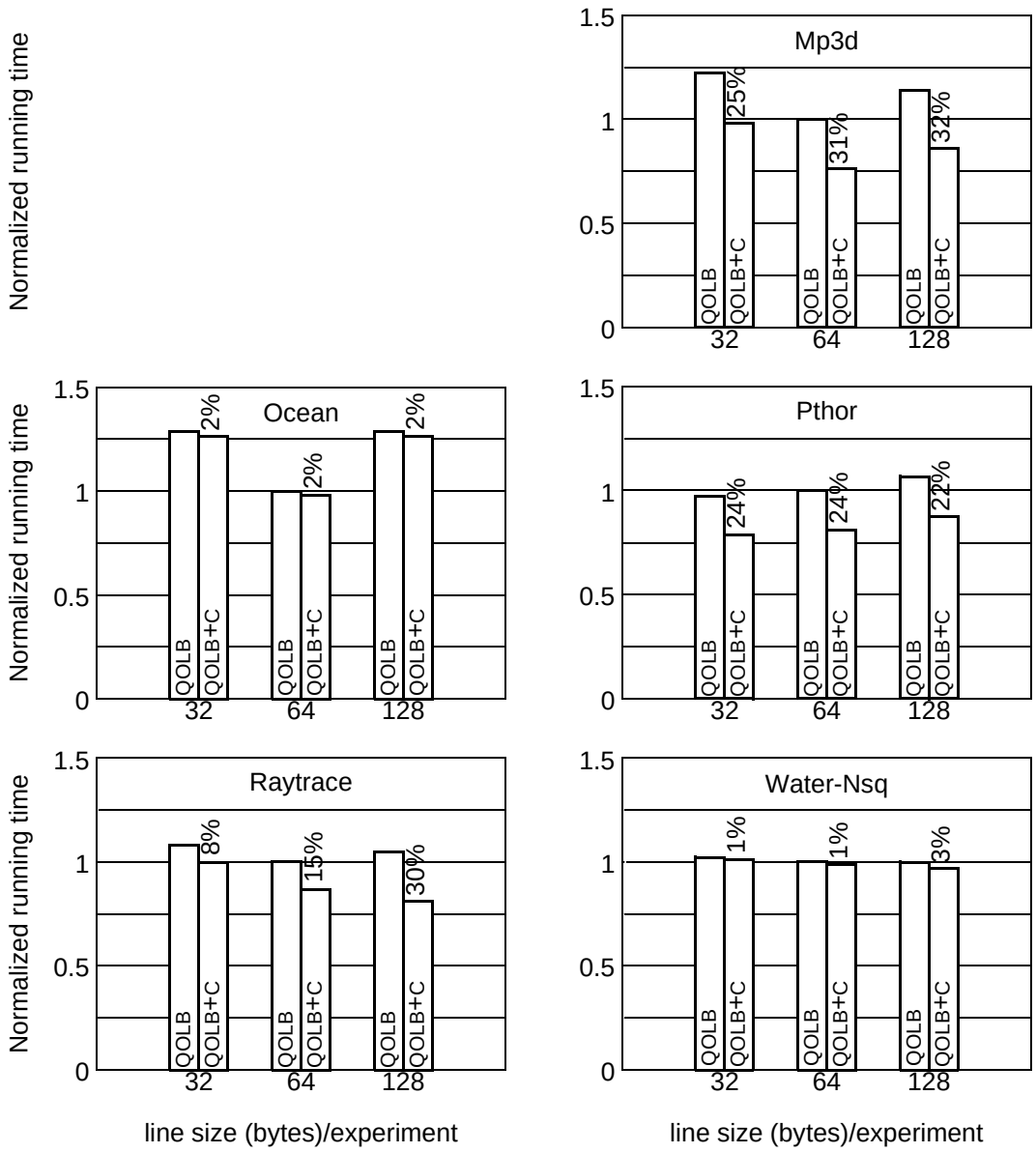



**Figure 5.1 Performance of collocation versus line size.** This figure shows the running time of each application's parallel section for three different line sizes (32, 64, and 128 bytes) and two experiments (QOLB and QOLB with the collocation mechanism enabled). For each application, the running time shown is normalized to that of the run with 64-byte line using QOLB without collocation. The percentage displayed on top of the right bar of each bar pair shows the speedup achieved by collocation for that particular run.

These results show that for at least some benchmarks, the choice of the block size can have an impact on the performance of collocation.

## 5.3 A new collocation strategy: VAQUM

The collocation strategies proposed so far suffer from either not bringing locally critical data early enough (prefetching scheme) or not being able to handle arbitrarily sized critical data (cache-coherence-based scheme). Ideally, collocation should handle *all* the critical data and transfer these data directly with the lock. I propose a new synchronization primitive, dubbed VAQUM, that attempts to overcome the limitations of previous schemes and to approach the ideal of collocation.

VAQUM is an extension to QOLB; it implements all the four synchronization mechanisms (local spinning, queue-based locking, collocation, and synchronous prefetch) that are described in Chapter 3; therefore, VAQUM will perform at least as well as QOLB. The novel aspect of VAQUM is that it has the ability, based on input from the user, to optimize the transfer of the lock and data. The optimization takes two forms. First, VAQUM transfers all the data that the user has specified as belonging to a particular lock. Second, VAQUM selects the type of message most appropriate to transfer the data as rapidly as possible. Typically, the network fabric transports smaller messages faster than larger ones, as illustrated by Figure 5.2, which plots message round-trip time as a function of message size for the cluster of workstations described in Chapter 2. Note, however, that while a smaller message minimizes network latency, it typically cannot achieve peak bandwidth at the same time. Achieving peak bandwidth usually requires large

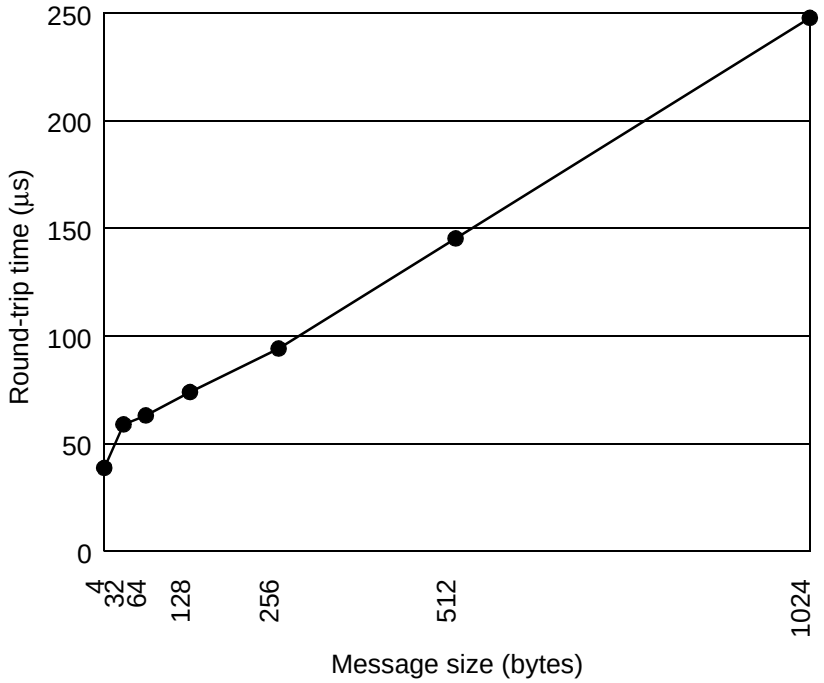



**Figure 5.2 Message latency versus message size.** This figure plots the round-trip time for messages of different size. This graph is a reproduction of data that appear originally in Figures 3-13 and 3-14 of Schoinas' thesis [Sch97]. Schoinas measured these latencies on the same cluster of workstations as the one used in this thesis and for which a description appears in Chapter 2. To generate this data, Schoinas measured the round-trip latency of messages injected into and retrieved from the network using the Berkeley active message library [vECGS92].

messages. Choosing a message type most appropriate to transfer lock and associated data may improve synchronization performance. For instance, consider a lock that protects only one word. Using QOLB and assuming a cache line size of 128 bytes, the lock transfer time would be related to the round-trip time of a 128-byte message (74µs). On the other hand VAQUM would select the smallest message that fit the lock and the single word; for instance a 32-byte message, the round-trip delay of which is 59µs, or an improvement of 25% over the 128-byte message.

Of course, VAQUM cannot optimize the data transfer unless the programmer explicitly describes the location of all the critical data. If the programmer does not give any information, VAQUM will behave exactly as QOLB, benefiting from collocation if data happens to reside in the same cache line as the lock.

Another alternative to improving the performance of collocation consists of an hybrid solution that collocates as much data as possible in a cache line of fixed size and prefetches the rest at the beginning of a critical section. This strategy is easier to implement than VAQUM; however, it may not perform as well because (1) transfer of lock and data may not be optimized as well as in the case of VAQUM, and (2) the fraction of data that does not fit may not arrive early enough to avoid stalls inside the critical section. I do not study this technique further.

## 5.4 CLEAN

CLEAN is an extension of SOFTQOLB, the implementation of QOLB described in the previous chapter. CLEAN extends SOFTQOLB by allowing the user to specify for a given region of memory the granularity at which data consistency is maintained. This flexibility allows the user to select the coherence grain size most appropriate for the amount of spatial locality that a given data structure possesses. If a data structure has little spatial locality, the user should allocate the structure in a region of memory using a smaller coherence grain size. Alternatively, if a data structure displays more spatial locality, the program should choose a larger grain size. This user-selectable coherence grain size, coupled with CLEAN's coherence protocol that supports the QOLB synchronization primitive,

**Table 5.1 Summary of the CLEAN interface.**

| FUNCTION | DESCRIPTION | ARGUMENT |
|---|---|---|
| v_allocate | Allocate an object | Size of object in bytes |
| v_deallocate | Deallocate an object | Pointer returned by v_allocate |
| v_enter_exclusive | Start exclusive object access | Pointer returned by v_allocate |
| v_exit_exclusive | End exclusive object access | Pointer returned by v_allocate |
| v_prefetch_exclusive | Prefetch object for exclusive access | Pointer returned by v_allocate |
| v_prefetch_shared | Prefetch object for exclusive access | Pointer returned by v_allocate |
| v_flush | Flush object from cache | Pointer returned by v_allocate |

suggests a possible implementation of VAQUM. In summary, CLEAN combines the fine-grain replication of shared data with the efficient access to exclusively held data. Table 5.1 summarizes the interface of the CLEAN system. The pointer returned by `v_allocate` is a pointer that conventional load and store instructions can use. For orthogonality's sake, the interface should also define `v_enter_shared` and `v_exit_shared`; however, in CLEAN, these functions would be null functions, since objects can be accessed directly through conventional load and store instructions to shared memory. Yet, a program can prefetch a shared object that it plans on accessing with `v_prefetch_shared`.

CLEAN runs on the cluster of dual-processor Sun SPARCstation 20s described in Chapter 2. By default, CLEAN allocates data in the region of memory managed at a granularity of 128 bytes. If otherwise specified, CLEAN chooses the region of memory with the smallest cache line size that can contain the amount of data that the program wants to allocate. Possible cache line sizes are all powers of two ranging from 32 to 2048 bytes.

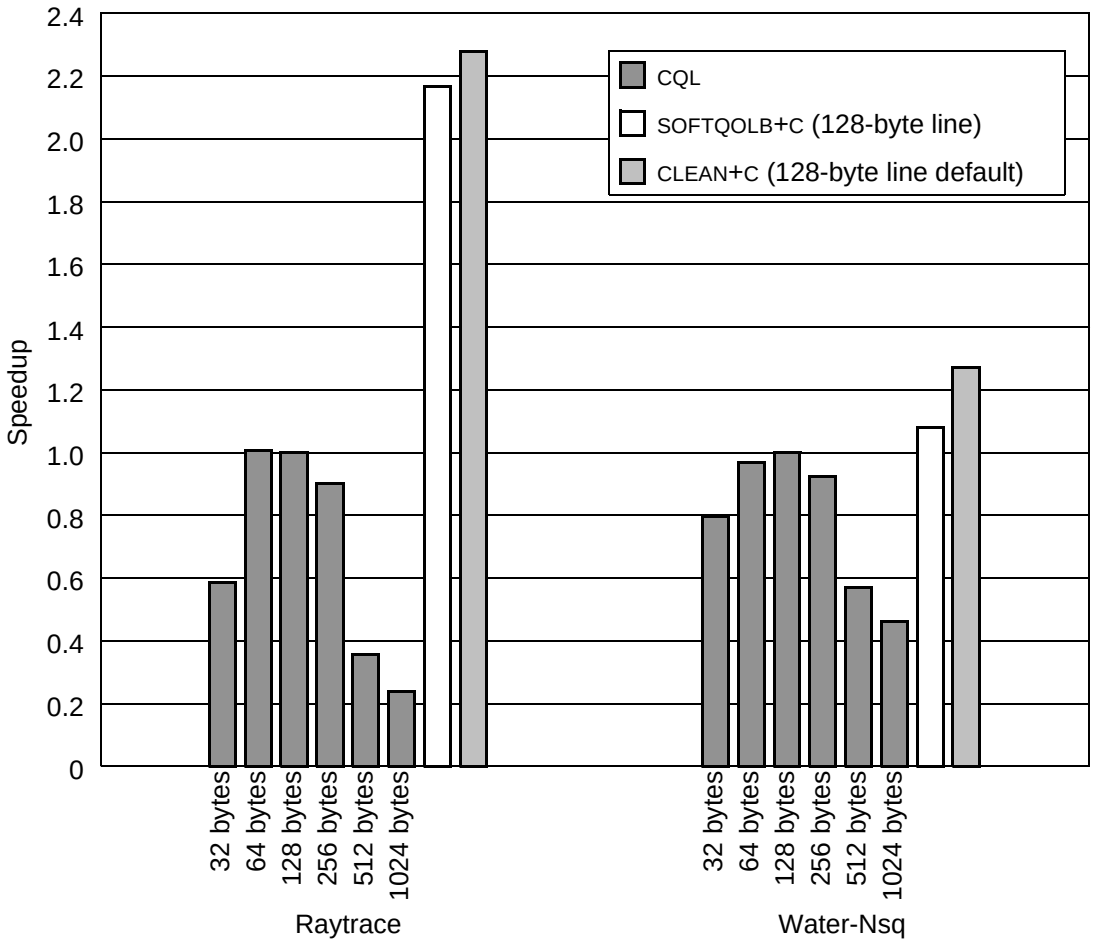



**Figure 5.3 CLEAN results.**

## 5.5 Results

I measure the performance of CLEAN, an implementation of VAQUM, for two benchmarks (Raytrace and Water-Nsq) and compare its performance with SOFTQOLB (with a constant coherence grain size of 128 bytes), and the centralized queued-lock (CQL) with coherence maintained at varying grain sizes (from 32 to 1,024 bytes).

The choices of coherence grain sizes are as follows. For Raytrace, I allocate all the locks and associated data in the memory region managed with 32-byte cache lines. I make Raytrace's own memory allocator allocate data in memory managed with 256-byte cache lines. Raytrace uses this allocator to allocate memory for the scene and ray data structures. All the other data is allocated in default memory. For Water-Nsq, I allocate the molecule data structure in 512-byte memory and the other locks in 32-byte memory. The remaining data is allocated in default memory.

Figure 5.3 plots speedups over the base case of CQL with a coherence grain size of 128 bytes. All runs use eight nodes. First, it is interesting to note that for these two benchmarks, the grain size of 128 bytes is a good choice since no other choice exceeds the base case performance by more than 1%. This result corroborates results stated elsewhere [Sch97]. Also, SOFTQOLB and VAQUM always outperform CQL no matter the grain size at which the protocol maintains coherence. Finally, VAQUM outperforms SOFTQOLB by 5% for Raytrace and 18% for Water-Nsq.

## 5.6 Related work

Bitar and Despain first discuss a cache coherence protocol that implements collocation [BD86]. In their scheme, collocation is achieved by storing data in the same cache line that stores the lock. Goodman, Vernon, and Woest discuss a similar collocation strategy in a synchronization primitive called QOLB [GVW89]. Both of these proposals are limited in the amount of data that can be successfully collo-

cated; these proposals are constrained by the cache line size of their underlying coherence protocols.

Trancoso and Torrellas propose to use prefetch to simulate the behavior of collocation. Their technique inserts instructions at the beginning of each critical section to prefetch the data that are going to be accessed. Their technique is not constrained by the cache line size used by the system. However, this technique may not be able to issue prefetches soon enough to avoid altogether stalls due to accesses performed inside a critical section.

Many other systems share CLEAN's shared-memory interface centered around the declaration of objects that an application reads or accesses exclusively. These systems include CRL [JKW95], Shared Regions [SGZ93], and Cid [Nik94]. These systems have in common that program annotations are required before accesses to objects can take place correctly. In contrast, CLEAN does not enforce the use of the annotations it defines;[1] instead it views these annotations as hints to achieve higher performance through cooperation between the program and the underlying system. The Midway distributed shared-memory system [BZS93] is perhaps most similar to CLEAN. Both share the optional program annotations to improve the program's performance. However, CLEAN supports a more efficient synchronization primitive that can transfer a lock in one network transaction and prefetch protected data ahead of time. Wood and his colleagues also propose a cooperative framework with CICO [HLRW92, HLRW93, WCF+93, LCW94], a performance

1. However, delineating the critical sections is still required.

model to help users understand the delays associated with different events that may occur in a distributed shared-memory system.

For each region of memory, CLEAN lets the programmer specify the best granularity to use by the underlying cache coherence protocol. This specification is performed at compile time or when memory allocation is performed and is typically set for the duration of the application's execution. In contrast, Dubnicki and LeBlanc [DL92] discuss a coherence scheme that selects the best grain size dynamically. The advantage of their approach is that their scheme can adapt with changing program behavior; however this flexibility comes at the cost of a more complex protocol.

## 5.7 Summary

This section focused on studying more carefully the performance implications of alternative implementations of collocation. The purpose of collocation is to eliminate the read and write misses associated with accesses to critical data while executing a critical section. I argued that read or write misses caused by accesses to protected data while executing in a critical section are an integral part of inefficiencies of synchronization. Therefore, collocation has the potential to reduce synchronization inefficiencies dramatically, thereby shortening the synchronization period, increasing locking throughput, and decreasing lock contention (the lock not being held as long).

Collocation will become even more important since it helps reduce or even eliminate global communication, which is becoming relatively slower with respect to ever faster processors.

I discussed two known methods to approach the ideal of collocation. The first scheme, due to Trancoso and Torrellas [TT96], inserts instructions at the critical section beginning to prefetch protected data. The other scheme uses the "prefetching" ability of a cache line. Both of these schemes suffer from drawbacks. The first scheme may not bring data where it is needed soon enough; while the second scheme cannot handle protected data of arbitrary size.

I proposed VAQUM, an extension to QOLB, that can handle critical data of arbitrary size. VAQUM performs at least as well as QOLB; and VAQUM has the potential to outperform QOLB if the programmer provides information about the nature of the protected data. With the appropriate information, VAQUM will attempt to optimize the transfer of lock and *all* associated data.

I described and evaluated CLEAN an implementation of VAQUM on a cluster of workstations. CLEAN is implemented as an extension to the SOFTQOLB primitive described in the previous chapter. CLEAN allows the user to specify the granularity at which data consistency is maintained in a given region of memory. This flexibility coupled with SOFTQOLB, the efficient synchronization primitive, demonstrated a possible implementation of VAQUM. For a couple of benchmarks I compared the performance of CLEAN to the performance of CQL with varying cache line sizes and to the performance of SOFTQOLB with a cache line size of 128 bytes.

For these particular cases, CLEAN always outperformed the alternatives, however not by much.

# Chapter 6

# Conclusion

## 6.1 Thesis summary

Efficient synchronization primitives are essential for achieving high performance in fine-grain, shared-memory parallel programs. One function of synchronization primitives is to enable exclusive access to shared data and critical sections of code.

Instead of focusing on the individual latencies associated with mutually exclusive accesses to critical section, I focused on the global throughput of critical section accesses. I defined the notion of a *synchronization period*: one "cycle" of multiple serialized accesses to critical section. I broke this period into three phases (*Transfer, Load/compute,* and *Release*), and classified the components of each of these phases as either unavoidable latencies or removable inefficiencies.

I identified four optimizing mechanisms (*local spinning, queue-based locking, collocation,* and *synchronous prefetch*) that can assist in eliminating the inefficiencies of critical section accesses. Helped with the synchronization period

framework, I discussed how each mechanism contributes to improve the performance of synchronization.

I performed a thorough evaluation of this space, simulating the performance of 16 locking constructs (formed from six base primitives: test&set, test&test&set, MCS, LH, M, and QOLB) in detail with both real parallel applications and the more traditional microbenchmarks.

The results showed that local spinning consistently aids performance, but not by very much. Queue-based locking is the most effective synchronization mechanism, except in the cases where the overhead of MCS, LH, or M locks hurt low-contention critical section access latencies. With test&set and test&test&set, collocation of the lock and protected data in the same cache line showed wildly different effects depending on the benchmark analyzed; collocation may greatly increase or decrease performance. Collocation can hurt performance when there is heavy lock contention or when a lock is held a long period of time giving other processors a chance to steal data from the processor in the critical section. However, with exponential back-off added, test&test&set with collocation performed as well as QOLB in nearly all cases. Exponential back-off helped overcome the impact of lock contention very well. Collocation consistently improved the performance of QOLB. Synchronous prefetching is the least effective of any of the mechanisms, although unexplored compiler techniques might show greater promise.

I discussed how to implement efficient synchronization that incorporates all the four synchronization mechanism. I identified six hardware mechanisms (*naming*, *protocol processing*, *synchronous cache-to-cache data transfer*, *placeholder alloca-*

*tion*, *non-blocking instructions*, and *association of a lock and data*) required to support efficient synchronization in a shared-memory multiprocessor. I described several alternatives to implement each of these hardware mechanisms and described two complete systems that represent two extremes in the cost/performance spectrum. A high-end system represents an ideal implementation of comparable performance to the system that my detailed simulator models. It requires extension of the instruction set architecture (ISA) and integrated large on-chip coherent cache with a hard-wired protocol that implements a QOLB-like locking primitive. At the other extreme, there is an all-software system that manages parts of local memory as a large fully associative cache. I described an implementation of such a solution on a cluster of 16 workstations. This implementation supports the efficient QOLB protocol and I evaluated its performance.

The results showed that a low-cost implementation of an efficient synchronization primitive is possible; however the all-software implementation of the hardware mechanisms required to support QOLB introduces inefficiencies that prevent QOLB from reaching its full potential. In a limited set of experiments, I demonstrated that this low-cost implementation of QOLB is never slower than 25% compared to the other alternatives, and for some benchmark this implementation is substantially better than the alternatives.

Finally, I discussed the limitation of a typical implementation of collocation which takes advantage of the fixed grain size at which coherence is maintained. I proposed a new synchronization primitive, VAQUM, that overcomes this limitation. It is a primitive that uses, but does not require, user annotations to transfer

all the data associated with a lock as efficiently as allowed by the underlying messaging layer. I described a new distributed shared-memory system called CLEAN that implements VAQUM. CLEAN defines the notion of objects of arbitrary size that live in shared memory. Thus, programs can access these objects using conventional load and store instructions, which, if they miss in local memory, trigger the proper coherence actions. In addition, CLEAN supports VAQUM providing programs with very efficient exclusive access to these objects. In effect, for each data structure defined in a program, the user can specify the best suited grain size at which to maintain coherence. CLEAN runs on a cluster of workstations and I evaluated its performance.

The results showed that CLEAN is at least as fast as SOFTQOLB and that CLEAN achieves speedups that are reasonable even for challenging applications and small problem sizes.

An important result of this thesis is the consistent large performance gain that QOLB achieves if implemented in hardware. Collocation increases this gain even further. Graunke and Thakkar [GT90] concluded that "... elaborate hardware [synchronization] schemes are unnecessary even when considering larger non-bus-based [systems]." Mellor-Crummey and Scott stated [MCS91b] that "special purpose synchronization mechanisms, such as QOLB, are unlikely to outperform our MCS lock by more than 30%." The results presented here refute these assertions; QOLB outperforms MCS by 40% for Mp3d.

Lim and Agarwal claimed [LA94] that reactive synchronization "reduces the motivation for providing hardware support for queue locks." Since QOLB outper-

forms the best software locks under either low- or high-contention conditions, it should also outperform reactive synchronization schemes. The results presented here confirm this hypothesis—QOLB speedups are from 6% to 97% higher than reactive synchronization, and this disparity only increases by adding collocation and synchronous prefetch to QOLB.

## 6.2 Future directions

### 6.2.1 Synchronization primitives and out-of-order execution

I have analyzed the impact of some out-of-order execution techniques on the performance of synchronization primitives. Due to limitations of my simulation environment, I have analyzed the performance of synchronization primitive with the presence of very aggressive write buffers, but I have not studied the impact that non-blocking load instructions and speculative execution might have on synchronization performance. Tools [PRA97, RHWG95] have recently appeared that would make such an evaluation possible.

### 6.2.2 Synchronization performance in non-scientific workloads

This thesis used only benchmarks from Stanford's SPLASH and SPLASH-2 benchmark suites containing mostly scientific applications. Two application domains that have emerged as the most important users of parallel shared-memory platforms are commercial and transactional workloads. The performance of synchronization primitives needs analysis in the context of these popular application domains. Also, to my knowledge there is no existing study that investigates the

impact of efficient synchronization support on the performance of operating systems managing resources in a parallel system.

### 6.2.3 Wait-free synchronization

Jensen, Hagensen, and Broughton proposed load-reserve and store-conditional instructions as a new solution to provide ISA support for synchronization [JHB87]. So far the application of these instructions has been restricted to simulating other well-known synchronization primitives such as test&set, swap, and compare&swap, the usefulness and performance of which are well understood. However, much less understood is the performance behavior of wait-free objects [Her91], the construction of which is possible with an ideal implementation of these instructions. With the advent of more interesting use of these instructions (see for instance the interesting work by Moir [MA95, Moi98, Moi97]), it would be interesting to compare wait-free algorithm implementations with their more conventional counterparts. Theory suggests that there might be a gap between these two types of implementations [ALS94].

### 6.2.4 Unification of speculative execution and wait-free synchronization

Herlihy and Moss [HM93], and Stone and her colleagues [SSHT93] independently proposed an extension, called transactional memory, to the load-reserve and store-conditional instructions that support multiple reservations instead of a single one. This extension essentially treats a critical section as a transaction the execution of which can succeed or fail; if a processor detects failure, it typically restarts the atomic sequence. Support for this extension requires special hard-

ware that may prove to be prohibitive. However, considering that aggressive processor implementations are already supporting speculative write operations, speculative state might soon encompass the entire first level cache. With such support for extremely aggressive speculative execution in place, I believe a trivial extension to the speculative first level cache would provide the desired functionality of transactional memory described earlier. Updates performed inside the critical section would remain “speculative” until the program finishes executing the critical section, at which point the program would check for violation of atomicity. If there is no violation, the program commits the speculative state; the program can restart the atomic sequence otherwise. Speculative first level caches would improve both single thread performance through extremely aggressive speculation and parallel program performance through non-blocking, opportunistic execution of critical sections.

# Appendix A

## SOFTQOLB

SOFTQOLB is a cache coherence protocol that combines a novel replication policy proposed by Reinhardt, Larus, and Wood in Stache [RLW94] with QOLB, an efficient synchronization primitive proposed by Goodman, Vernon, and Woest [GVW89]. Stache is an implementation of user-level transparent shared-memory defined with Tempest [RLW94], an interface that exposes low-level communication and memory-system mechanisms so programmers and compilers can customize policies for a given parallel application. Specifically, Stache manages part of a processor's local memory as a large, fully-associative cache for remote data. With such an unusually large and flexible cache, Stache eliminates the capacity and conflict misses caused by applications' working sets that are too large to fit in smaller hardware caches with limited associativity. Furthermore, Stache has the advantage over conventional directory-based cache coherence protocol not to have to return a cache line to the home node upon cache replacement. Stache shares this characteristic with COMA memory systems [Hag92] but without the draw-

backs of their implementation costs and complexity. QOLB is an efficient synchronization primitive that implements all the four locking mechanisms listed in Chapter 3. These mechanisms are local spinning, queue-based locking, collocation, and synchronous prefetch.

I have designed, verified with an automatic protocol verifier [DDHY92], and implemented SOFTQOLB, a cache coherence protocol that combines the fine-grain replication support offered by Stache with the efficient support for synchronization afforded by QOLB. In what follows, I describe SOFTQOLB following closely the style adopted by Hennessy and Patterson to describe a directory protocol in Chapter 8 of the second edition of their book “Computer Architecture: A Quantitative Approach” [HP95]. First, I present the part of SOFTQOLB that implements Stache, followed by its QOLB component, and, finally, I conclude with a description of the interaction between these two modes of operations.

The basic principle that guided the design of Stache is that whenever possible local memory should satisfy local accesses to shield the processor from the long network latencies. Stache fulfills this principle in one of two ways. Either local memory stores the memory location, or Stache arranges to store a copy of that location in the portion of local memory that Stache manages as a cache.

Stache associates a directory with each memory module that stores shared data. The directory keeps track of the remote nodes that store a copy of a memory location and the states in which the remote nodes store these copies. The directory entry associated with a memory location is in one of the following states:

- *Idle*—No remote node has a copy of the memory location and the local processor has read and write access to that memory location.
- *Shared*—One or more remote nodes have a read-only copy of the memory location, the value at the memory is up to date, and the local processor has read-only access to that memory location.
- *Exclusive*—Exactly one remote node has a read/write copy of the memory location, the value at the memory is not up to date, and the local processor has no access privilege to that memory location.

In addition to states, the directory keeps a list of the remote nodes that have a copy of the memory location. In this particular implementation, the directory stores that information as explicit pointers when the sharing set contains four or less elements, or as a bit vector otherwise.

The state of a memory location at a remote node is one of the following:

- *Invalid*—The remote node does not possess a copy of that memory location.
- *Shared*—The remote node has a read-only copy of that memory location.
- *Exclusive*—The remote node has the only valid copy of that memory location and the node has read and write access to the copy.

Stache assumes that the processors block on a memory access that misses until it completes. Furthermore, Stache (and SOFTQOLB) assumes a memory model that satisfies the constraints of sequential consistency.

Table A.1 shows the message types that the protocol sends among the caches and the directories. *Local* node is the node where the request originates. *Home* node is the node where the memory location and associated directory information

of an address reside. The *remote* node is a node which has a copy of a memory location, shared or exclusive.

Figure A.1 depicts the Stache cache coherence protocol for state transitions occurring on the cache side; while Figure A.2 depicts the transitions occurring on the home side. These state diagrams do not represent all the details of Stache, but they give enough information to understand the workings of this protocol and to estimate its implementation complexity.

Figure A.1 (a) and Figure A.2 (a) represent state transitions caused by local requests; while Figure A.1 (b) and Figure A.2 (b) represent transitions caused by remote requests. Next to the arcs, the figures show in regular type the stimuli that cause the transitions; the figures show the actions taken on the transitions in bold.

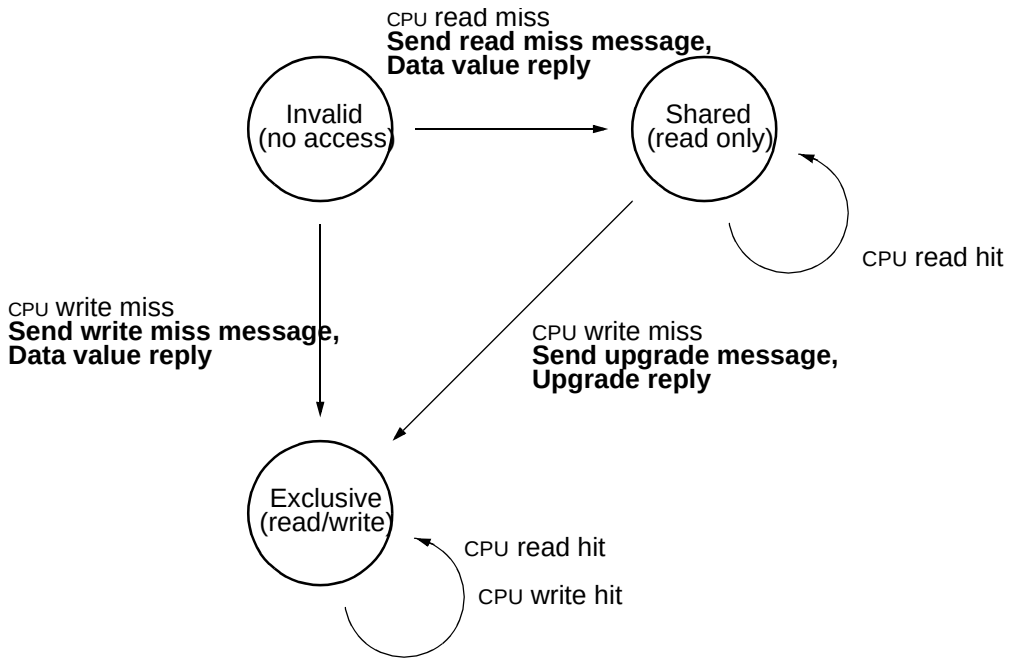
On the cache side, one of two local events (read or write miss) causes Stache to send a request (read miss, write miss, or upgrade) to the directory and to wait for a reply (data value or upgrade reply). Stache processes a remote request (invalidate or write-back which always originate from home) sent to a cache immediately and sends the appropriate reply back home.

On the directory side, Stache handles local requests (read or write miss) in a manner similar to the cache-side protocol. In contrast, remote requests to the directory may require more processing than the protocol transitions we have seen so far. Indeed, Stache may need, upon receiving a remote request (read miss, write miss, or upgrade), not only to change state but also to send additional mes-

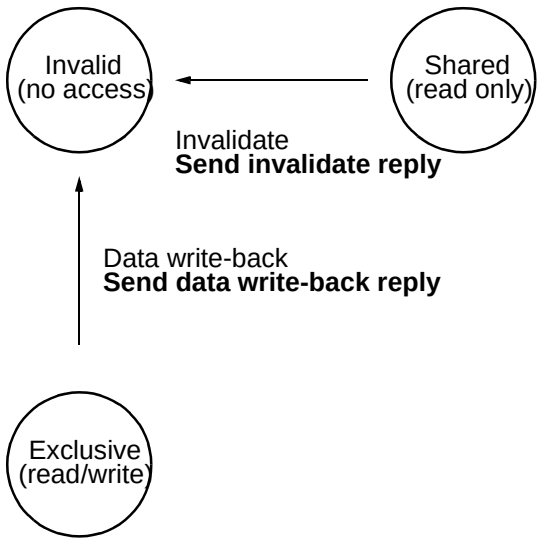
**Table A.1 The possible messages sent among nodes to support Stache.** The first three messages are miss requests sent by the local cache to home. The fourth and fifth messages are requests sent to a remote cache by home when home needs to satisfy a read miss, or write miss, or upgrade request. Stache uses data value replies to send a value from the home node back to the requesting node. Data value write-backs occur when replying to write-back messages from home. Writing back the data value whenever the line becomes shared simplifies the number of states in the protocol since any dirty line must be exclusive and any shared line is available at the home memory. The last two messages are replies to upgrade and invalidate requests respectively.

| MESSAGE TYPE | SOURCE | DESTINATION | MESSAGE CONTENTS | MESSAGE FUNCTION |
|---|---|---|---|---|
| Read miss | Local cache | Home directory | P, A | Cache does not have a readable copy of memory location A to satisfy processor P's read access; request data and make P a read sharer. |
| Write miss | Local cache | Home directory | P, A | Cache does not have a writable copy of memory location A to satisfy processor P's write access; request data and make P the exclusive owner. |
| Upgrade | Local cache | Home directory | P, A | Cache has a readable, but not writable, copy of memory location A and cannot satisfy processor P's write access; request upgrade to exclusive state. |
| Invalidate | Home directory | Remote caches | A | Invalidate a shared copy of data at address A. |
| Write-back | Home directory | Remote cache | A | Send cache line at address A back to its home; invalidate the line in the cache. |
| Data value | Home directory | Local cache | Data | Return a data value from the home memory. |
| Data write-back | Remote cache | Home directory | Data | Write back a data value to its home. |
| Upgrade reply | Home directory | Local cache | | Inform the local cache that Stache has processed the upgrade request. |
| Invalidate reply | Local cache | Home directory | A | Inform home that the local cache has invalidated the cache line at address A. |

sages. In that case, Stache must then wait until the responses come back to reply to the original requester. Also Stache must take great care to handle the case where two remote nodes request an upgrade simultaneously.




(a) State transitions based on requests from CPU




(b) State transitions based on remote requests

**Figure A.1 Cache-side state transitions for the Stache cache coherence protocol.** The top diagram shows transitions based on requests issued by the local processor and the bottom diagram shows transitions based on remote requests. A circle represents a state with, displayed inside, the name of the state and, in parenthesis, the types of local accesses permitted without a state change. An arrow connecting two states represents a valid state transition. Shown next to each arc in regular type is the stimulus that causes the transition;

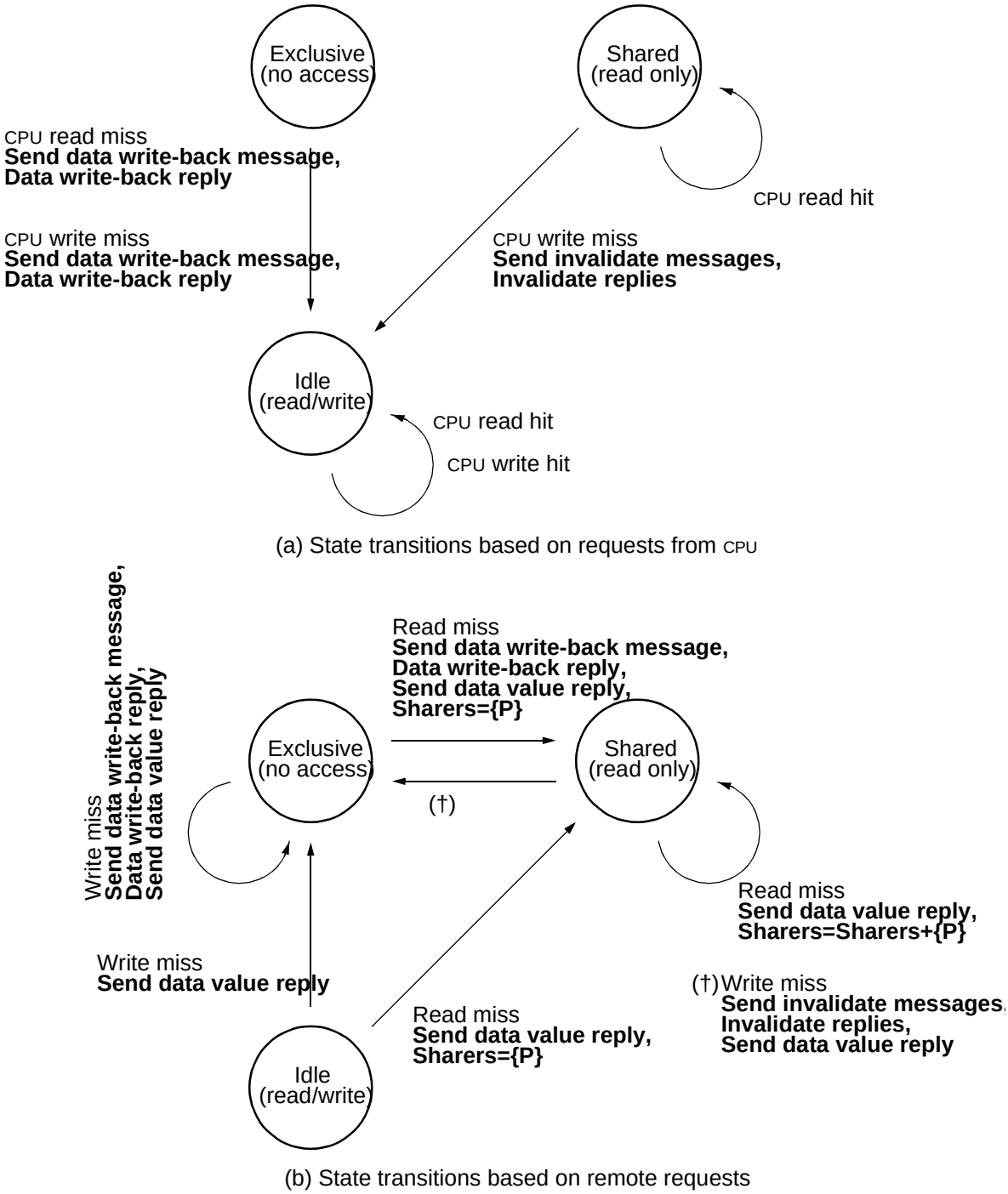



**Figure A.2 Home-side state transitions for the Stache cache coherence protocol.** This figure uses the same conventions as those described in Figure A.1. P is a field in the message that identifies the requester.

QOLB builds a queue of processors waiting to acquire a lock. A processor releases a lock by issuing a `deqolb` instruction with the address of the lock. This instruction causes QOLB to ship the corresponding cache line to the next processor in line, if any. A processor requests a lock by issuing an `enqolb` instruction. Both of these instructions are non-blocking; in other words, the processor can issue other instructions while the QOLB processing occurs concurrently. A processor will know that it is safe to proceed with the execution of the critical section, when the `enqolb` instruction returns with a status indicating that the corresponding cache line is currently available locally in locked state.

SOFTQOLB implements the QOLB queue as a linked list where each node maintains a pointer to the next node in line. In addition, the directory maintains a pointer to the current insertion point (i.e., the tail of the queue). A node wishing to enter the queue, first requests the identity of the tail node, then becomes the tail node itself, and, finally, updates the old tail node's next pointer by sending it a queueing message.

During QOLB operations, a directory entry is in one of the following states:

- *Idle*—There is no queue, the memory location is at home, and the local processor has read and write access to that memory location.
- *Locked*—The queue is non-empty, the home node is not in the queue, and the local processor has no access privilege to that memory location.
- *Owned*—The home node has the only copy of the memory location, that copy is locked, home has read and write access to that memory location, and home

will not give away that memory location until the local processor issues a `deqolb` instruction.

- *Needed*—Same as owned, except that the queue contains at least two elements.
- *Tail*—The home node is the last one (possibly temporarily) to have requested to receive the memory location locked, home has allocated space to store a copy of that memory location (shadow cache line), and home has no access privilege to that memory location.
- *Middle*—Same as tail, except that at least one other node has requested the line after the local processor did.

The states for the cache side are very similar:

- *Invalid*—The remote node does not have a copy of the memory location and has no access privilege to it.
- *Owned*—The remote node has the memory location locked, the queue is otherwise empty, and the remote node has read and write access to that memory location.
- *Needed*—Same as owned, except that the queue contains at least two elements.
- *Tail*—The remote node has requested a copy of the memory location and has no access privilege to it.
- *Middle*—Same as tail, but at least one other node has requested the line after the local processor did.

Table A.2 shows the message types required to support QOLB in the SOFTQOLB protocol. In addition to fields identifying the requester (P) and the address being

**Table A.2 The possible messages sent among nodes to support QOLB.** The first message is a request sent by the local cache to home. The second message is a message sent by home to indicate a cache requester the current queue insertion point. The third message allows the recipient to update its next pointer correctly. The fourth message sends the data value to the next processor in line. The last three messages are replies to QOLB, queue, and data value messages, respectively.

| MESSAGE TYPE | SOURCE | DESTINATION | MESSAGE CONTENTS | MESSAGE FUNCTION |
|---|---|---|---|---|
| QOLB | Local cache | Home directory | P, A | Processor P issued an `enqolb` instruction at address A; request identity of current tail. |
| Tail | Home directory | Local cache | T, A | Send current tail identity; make requester (P) new tail. |
| Queue | Any node | Remote cache | P, A | Insert processor P into queue behind current tail; make P old tail's next node. |
| Data value | Any node | Any node | A, Data | Current owner issued a `deqolb` instruction; send data value to next processor in queue; invalidate local copy; make recipient new owner. |
| QOLB reply | Home directory | Local cache | T, A | Send current tail identity; make requester (P) new tail; insert P into queue behind home; make P home's next node. |
| Queue reply | Remote cache | Any node | A | Acknowledge new tail. |
| Data value reply | Any node | Any node | A | Acknowledge data value transfer. |

accessed (A), messages may contain a field identifying the current insertion of point in the queue. Note also, that, unlike Stache, some messages in Table A.2 are no longer confined to be exchanged exactly between a home node and a cache

node. For instance, the source and destination of a data value message can be any node.

The act of joining the queue upon executing the `enqolb` instruction consists of two steps: send home a QOLB message to determine the current location of the tail of the queue (the response comes in the tail reply) and, then, send tail a queue message to update the linked list correctly (SOFTQOLB has committed this operation when the requester has received the queue reply). An optimization not described in detail here can perform these operations in one step if home and tail happen to be the same node.

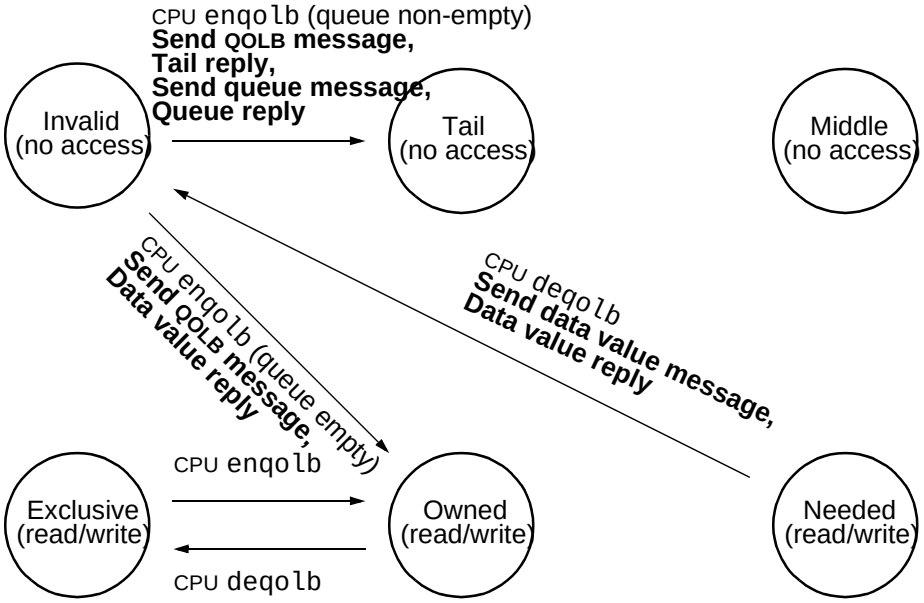
Figure A.3 and Figure A.4 depict the part of the SOFTQOLB coherence protocol that supports efficient synchronization (i.e., QOLB). Again, these state diagrams do not represent all the details of the protocol. In particular, these diagrams assume that the processor does not issue `deqolb` instructions while the memory location is in state tail or middle. I address this particular issue later when I describe the interaction between Stache and the synchronization support in SOFTQOLB.

Figure A.3 shows the cache-side state transitions; while Figure A.4 shows the home-side transitions. Figure A.3 (a) and Figure A.4 (a) represent state transitions caused by local requests (`enqolb` and `deqolb` instructions); while Figure A.3 (b) and Figure A.4 (b) represent transitions caused by remote requests.

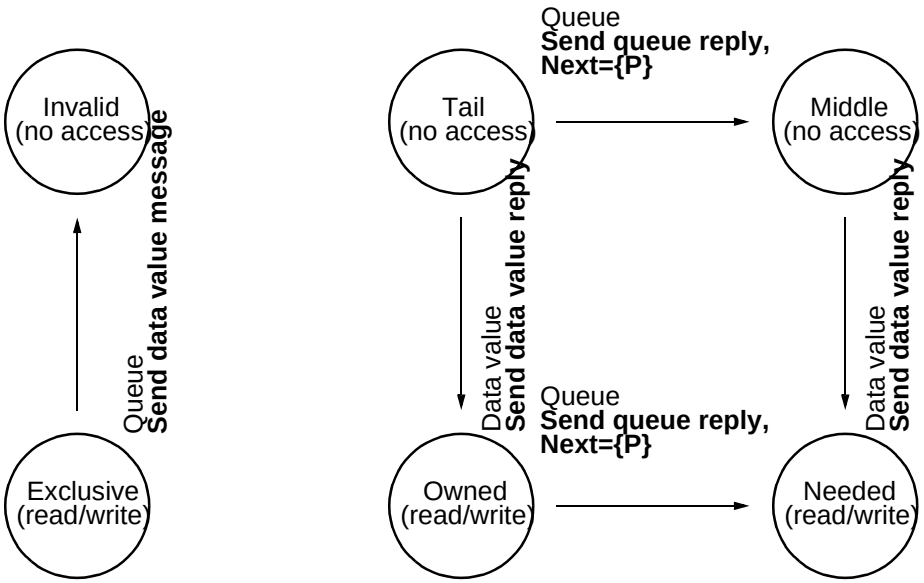
A node joins a queue by sending a QOLB message first, then a queue message. While waiting to obtain the cache line exclusively, the node may receive a queue request, which it honors by updating its next pointer and sending a queue reply.

A home node, in addition to supporting the operations just described, is also responsible to keep track of the current tail node, which represent the queue insertion point.

SOFTQOLB must expect `enqolb` and `deqolb` instructions at any time (recall that these instructions are non-blocking). Figure A.3 and Figure A.4 only show the basic transitions caused by `enqolb` or `deqolb` and disregard (for now) the question of dealing with `deqolb` instructions issued to states tail and middle. All other executions of `enqolb` of `deqolb` instructions not shown in Figure A.3 and Figure A.4 do not cause state transitions; SOFTQOLB ignores them.




(a) State transitions based on requests from CPU




(b) State transitions based on remote requests

**Figure A.3 Cache-side state transitions for the QOLB cache coherence protocol.** This figure uses the same conventions as those described in Figure A.1 and Figure A.2.

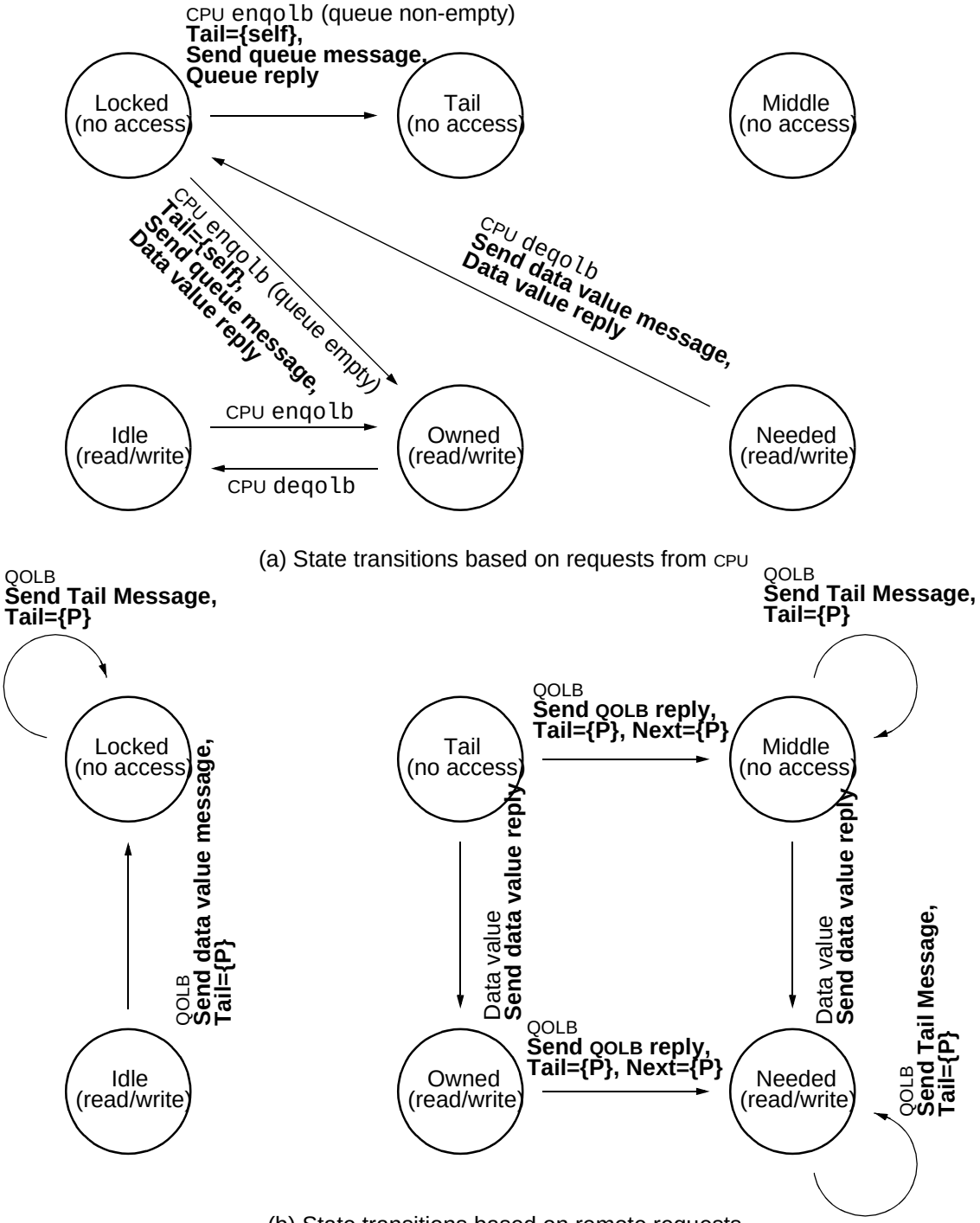



**Figure A.4 Home-side state transitions for the QOLB cache coherence protocol.** This figure uses the same conventions as those described in Figure A.1 and Figure A.2. Self denotes the node executing the `enqolb` instruction.

The descriptions so far have not considered the possibility of one or several nodes reading (or writing) a memory location concurrently with other nodes issuing QOLB instructions to the same location. Also, the SOFTQOLB description has not considered the possibility that a node might wish to undo the effect of an `enqolb` instruction.[1]

When a QOLB request finds the state at the directory to be either shared or exclusive, SOFTQOLB must first send invalidate messages or write-back request (as appropriate) to force the state at the directory back to idle. Then, the QOLB request proceeds as described earlier.

When a read or write miss finds that a QOLB queue has formed, it must first destroy the queue forcing the memory location back home. Then, the read or write request proceeds as described earlier. The home node collapses the queue on the behalf of the requester one node at a time starting from the tail. This operation is sequential in the number of nodes that have joined the queue: its emphasis is on correctness, not on efficiency. Note also, that this operation requires a doubly linked list. Hence, every node must keep track of its predecessor in the queue in addition to its successor.

Finally, to support speculative execution, SOFTQOLB must be able to undo the effect of the `enqolb` instruction. With a doubly linked list, the undoing of `enqolb` is trivial: send both neighbors each other's addresses.

With the possibility of many outstanding protocol transitions taking place concurrently arises the possibility of deadlocks. One invariant that SOFTQOLB

1. For instance, a processor could have issued an `enqolb` instruction misspeculatively.

observes to prevent deadlocks is to avoid storing and then forwarding a QOLB request. However, Stache does not follow this convention entirely, therefore the protocol must take great care when reserving resources.

# References

[ABB64] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2):87–101, April 1964.

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[Abr70] Norman Abramson. The ALOHA system—another alternative for computer communications. In *Proceedings of the AFIPS Fall Joint Computer Conference*, volume 37, pages 281–285, November 1970.

[AC89] Anant Agarwal and Mathews Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 396–406, May 1989.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton J. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[AGGW92] Nagi M. Aboulenein, James R. Goodman, Stein Gjessing, and Philip J. Woest. Hardware support for synchronization in the Scalable Coherent Interface (SCI). Technical Report CS-TR-92-1117, Computer Sciences Department, University of Wisconsin, Madison, WI, November 1992.

[AGGW94] Nagi M. Aboulenein, James R. Goodman, Stein Gjessing, and Philip J. Woest. Hardware support for synchronization in the Scalable Coherent Interface (SCI). In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 141–150, April 1994.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[ALS94] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the Association for Computing Machinery*, 41(4):725–763, July 1994.

[And89] Thomas E. Anderson. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II (software), pages 170–174, August 1989.

[And90] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[ASHAA97] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 204–215, February 1997.

[BBD+87] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfield, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, New York, NY, 1987.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Feldermann, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[BD86] Philip Bitar and Alvin M. Despain. Multiprocessor cache synchronization: Issues, innovations, evolution. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT-LCS-TR-516, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1991.

[BDFL96] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, and Kai Li. Protected, user-level DMA for the SHRIMP network interface. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 154–165, February 1996.

[BG95] Doug Burger and James R. Goodman. Simulation of the SCI transport layer on the Wisconsin Wind Tunnel. Technical Report CS-TR-95-1265, Computer Sciences Department, University of Wisconsin, Madison, WI, March 1995.

[BKT87] Bob Beck, Bob Kasten, and Shreekant Thakkar. VLSI assist for a multiprocessor. In *Proceedings of the Second Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 10–20, October 1987.

[BLA+94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[BLS+95] Philip Bohannon, Daniel Lieuwen, Avi Silberschatz, S. Sudarshan, and Jacques Gava. Recoverable user-level mutual exclusion. In *Proceedings of the seventh IEEE Symposium on Parallel and Distributed Processing*, pages 293–301, October 1995.

[BW95] Doug Burger and David A. Wood. Accuracy vs. performance in parallel simulation of interconnection networks. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 22–31, April 1995.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON)*, pages 528–537, February 1993.

[CB94] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, April 1994.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, April 1981.

[CP78] Richard P. Case and Andris Padegs. Architecture of the IBM System/370. *Communications of the ACM*, 21(1):73–96, January 1978.

[Cra93] Travis S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report UW-CSE-93-02-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, February 1993.

[CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[Cyp90] Cypress Semiconductor, San Jose, CA. *CY7C601 SPARC RISC User's Guide*, second edition, 1990.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1992 IEEE International Conference on Computer Design*, pages 522–525, October 1992.

[DL92] Cezary Dubnicki and Thomas J. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170–180, May 1992.

[FBS89] Alessandro Forin, Joseph Barrera, and Richard Sanzi. The shared memory server. In *Proceedings of the Winter 1989 USENIX Conference*, pages 229–243, January 1989.

[FG91] Eric Freudenthal and Allan Gottlieb. Process coordination with fetch-and-increment. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 260–268, April 1991.

[FP91] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–63, May 1991.

[GAG+92] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[GGK+83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU ultracomputer—designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[GHG+91] Anoop Gupta, John L. Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[GK81] Allan Gottlieb and Clyde P. Kruskal. Coordinating parallel processors: A partial unification. *Computer Architecture News*, 9(6):16–24, October 1981.

[Gle91] Andrew Glew. Synchronization primitive implementation including the bus abandonment lock. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1991.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GLR83] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[Goo83] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[GT90] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors. In *Proceedings of the Third Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[GW88] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, May 1988.

[GW92] Anoop Gupta and Wolf-Dietrich Weber. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[Gwe98] Linley Gwennap. Alpha 21364 to ease memory bottleneck. *Microprocessor Report*, 12(14):12–15, October 1998.

[Hag92] Erik Hagersten. *Toward Scalable Cache Only Memory Architectures*. PhD thesis, The Royal Institute of Technology (KTH), Stockhom, Sweden, October 1992. Also appears as Technical Report SICS Dissertation Series 08, Swedish Institute of Computer Science, Kista, Sweden, October 1992.

[Hei98] John Heinlein. *Optimized Multiprocessor Communication and Synchronization Using a Programmable Protocol Engine*. PhD thesis, Stanford University, Stanford, CA, March 1998.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[HGDG94] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1994.

[HKO+94] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[HLRW92] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1992.

[HLRW93] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[HP95] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, second edition, 1995.

[HS95] Chris Holt and Jaswinder Pal Singh. Hierarchical N-body methods on shared address space multiprocessors. In David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia J. Torczon, and Layne T. Watson, editors, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 313–318, February 1995.

[IEE93] Institute of Electrical and Electronics Engineers, New York, NY. *IEEE Standard for the Scalable Coherent Interface (SCI)*, August 1993. ANSI/IEEE Std 1596-1992.

[Int96] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 1: Specifications*, January 1996.

[JHB87] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, November 1987.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 213–228, December 1995. Also appears as Technical Memorandum MIT-LCS-TM-517, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1995.

[JMH97] Teresa L. Johnson, Matthew C. Merten, and Wen-mei W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 57–64, December 1997.

[JSS97] Dongming Jiang, Hongzhang Shan, and Jaswinder Pal Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the Sixth Symposium on Principles and Practice of Parallel Programming*, pages 217–229, June 1997.

[KABG95] Alain Kägi, Nagi Aboulenein, Doug Burger, and James R. Goodman. Techniques for reducing the overheads of shared-memory multiprocessing. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 11–20, July 1995.

[Kax98] Stefanos Kaxiras. *Identification and Optimization of Sharing Patterns for Scalable Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin, Madison, WI, 1998.

[KBG97] Alain Kägi, Doug Burger, and James R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 170–180, June 1997.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[KCK98] Chen-Chi Kuo, John Carter, and Ravindra Kuramkote. MP-LOCKs: Replacing hardware synchronization primitives with message passing. Technical Report UUCS-98-021, Department of Computer Science, University of Utah, Salt Lake City, UT, 1998.

[KCK99] Chen-Chi Kuo, John Carter, and Ravindra Kuramkote. MP-LOCKs: Replacing h/w synchronization primitives with message passing. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 284–288, January 1999.

[KCPT95] D. A. Koufaty, X. Chen, David K. Poulsen, and Josep Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 255–264, July 1995.

[KDL+93] David J. Kuck, Edward S. Davidson, Duncan H. Lawrie, Ahmed Sameh, Chuan-Qi Zhu, Alexander V. Veidenbaum, J. Konicek, Pen-Chung Yew, Kyle Gallivan, William Jalby, Harry A. G. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, David A. Padua, Rudolf Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner. The Cedar system and an initial performance study. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 213–223, May 1993.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, Upper Saddle River, NJ, 1992.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[KRS86] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 218–228, August 1986.

[KRS88] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, October 1988.

[KS93] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: A new dimension for Cray Research. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON)*, pages 176–182, February 1993.

[KSR91] Kendall Square Research Corporation, Cambridge, Massachusetts. *KSR1 Principles of Operation*, 1991.

[KW98] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, July 1998.

[LA94] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, October 1994.

[LAD+92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[LC96] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[LCW94] James R. Larus, Satish Chandra, and David A. Wood. CICO: A practical shared-memory programming performance model. In Tony Hey and Jeanne Ferrante, editors, *Portability and Performance for Parallel Processing*, chapter 5, pages 99–119. John Wiley & Sons, Chichester, United Kingdom, 1994.

[Lim95] Beng-Hong Lim. *Reactive Synchronization Algorithms for Multiprocessors*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, February 1995.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John L. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[LR90] Joonwon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27–37, May 1990.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[LT88] Tom Lovett and Shreekant Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume II (architecture), pages 303–310, August 1988.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[MA95] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

[MB76] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[MCS91a] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MCS91b] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[MFHW96] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[MK97] Evangelos P. Markatos and Manolis G. H. Katevenis. User-level DMA without operating system kernel modification. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 322–331, February 1997.

[MLH94] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report T94:07, Swedish Institute of Computer Science, Kista, Sweden, February 1994.

[Moi97] Mark Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, September 1997.

[Moi98] Mark Moir. Fast, long-lived renaming improved and simplified. *Science of Computer Programming*, 30(3):287–308, March 1998.

[MS95] Maged M. Michael and Michael L. Scott. Implementation of atomic primitives on distributed shared memory multiprocessors. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 222–231, January 1995.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, San Francisco, CA, second edition, May 1994.

[Nik94] Rishiyur S. Nikhil. Cid: A parallel, "shared-memory" C for distributed-memory machines. In Keshav Pingali, Utpal Banerjee, David Gelernter, Alex Nicolau, and David A. Padua, editors, *Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing*, pages 376–390, August 1994.

[OSG98] David R. O'Hallaron, Jonathan Richard Shewchuk, and Thomas Gross. Architectural implications of a family of irregular applications. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 80–89, February 1998.

[PBG+85] Gregory F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, Kevin P. McAuliffe, E. A. Melton, V. Alan Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[PN85] Gregory F. Pfister and V. Alan Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[PRA97] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM reference manual version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, Houston, TX, August 1997.

[PY94] David K. Poulsen and Pen-Chung Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume II (software), pages 276–280, August 1994.

[RCCT90] Randall D. Rettberg, William R. Crowther, Philip P. Carvey, and Raymond S. Tomlinson. The Monarch parallel processor hardware design. *IEEE Computer*, 23(4):18–30, April 1990.

[Rei96] Steven K. Reinhardt. *Mechanisms for Distributed Shared Memory*. PhD thesis, University of Wisconsin, Madison, WI, 1996.

[RHL+93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 48–60, May 1993.

[RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, 3(4):34–43, winter 1995. This journal has since been renamed *IEEE Concurrency*.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[ROS93] ROSS Technology, Austin, TX. *SPARC RISC User's Guide: HyperSPARC Edition*, third edition, September 1993.

[RPW96] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.

[RS84] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[RSG93] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[RSS+95] Umakishore Ramachandran, Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Ivan Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proceedings of the Supercomputing '95*, pages 1737–1775, December 1995.

[Rud81] Larry S. Rudolph. *Software Structures for Ultraparallel Computing.* PhD thesis, New York University, New York, NY, December 1981.

[Sch97] Ioannis T. Schoinas. *Fine Grain Distributed Shared Memory on Clusters of Workstations*. PhD thesis, University of Wisconsin, Madison, WI, 1997.

[Sco96] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[SFC91] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Corporation, Palo Alto Research Center, Palo Alto, CA, December 1991.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain control for distributed shared memory. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

[SGV92] Steven L. Scott, James R. Goodman, and Mary K. Vernon. Performance of the SCI ring. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 403–414, May 1992.

[SGZ93] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. The shared regions approach to software cache coherence on multiprocessors. In *Proceedings of the Fourth Symposium on Principles and Practice of Parallel Programming*, pages 229–238, May 1993. Also appears as Technical Report 277, Computer Systems Research Institute, University of Toronto, Canada, October 1992.

[Sit92] Richard L. Sites. Alpha AXP architecture. *Digital Technical Journal*, 4(4):19–34, 1992.

[SL94a] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 101–114, November 1994.

[SL94b] Daniel J. Scales and Monica S. Lam. An efficient shared memory layer for distributed memory machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, Stanford, CA, July 1994.

[Smi81] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In Tien F. Tao, editor, *Proceedings of the SPIE (Real-Time Signal Processing IV)*, volume 298, pages 241–248, August 1981.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[SSHT93] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel & Distributed Technology*, 1(4):58–71, November 1993. This journal has since been renamed *IEEE Concurrency*.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[TMC91] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.

[TT96] Pedro Trancoso and Josep Torrellas. The impact of speeding up critical sections with data prefetching and forwarding. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume III (software), pages 79–86, August 1996.

[UIT94] Teruo Utsumi, Masayuki Ikeda, and Moriyuki Takamura. Architecture of the VPP500 parallel supercomputer. In *Proceedings of the Supercomputing '94*, pages 478–487, November 1994.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[WCF+93] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for cooperative

shared memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–167, May 1993.

[Web65] Webster. *Webster's Seventh Dictionary*. G. & C. Merriam Company, Springfield, MA, 1965.

[WG91] Philip J. Woest and James R. Goodman. An analysis of synchronization mechanisms in shared-memory multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 152–165, April 1991.

[WGH+97] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki, and Winfried Wilcke. The Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 98–107, June 1997.

[WH95] David A. Wood and Mark D. Hill. Cost-effective parallel computing. *IEEE Computer*, 28(2):69–72, February 1995.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[ZCC94] Xiaodong Zhang, Robert Castañeda, and Elisa W. Chan. Spin-lock synchronization on the Butterfly and KSR1. *IEEE Parallel & Distributed Technology*, 2(1):51–63, spring 1994. This journal has since been renamed *IEEE Concurrency*.

[ZY87] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, 13(6):726–739, June 1987.