Reactive Synchronization Algorithms for Multiprocessors

by

Beng-Hong Lim

B.S., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1986

M.S., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1991

Submitted to the
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Signature of Author ____________________
Department of Electrical Engineering and Computer Science
October 28, 1994

Certified by ____________________
Anant Agarwal
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ____________________
Frederic R. Morgenthaler
Chairman, EECS Committee on Graduate Students

Reactive Synchronization Algorithms for Multiprocessors

by

Beng-Hong Lim

Submitted to the Department of Electrical Engineering and Computer Science
on October 28, 1994 in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy
in Electrical Engineering and Computer Science

ABSTRACT

Efficient synchronization algorithms are hard to design because their performance depends on run-time factors that are hard to predict. In particular, the designer has a choice of *protocols* to implement the synchronization operation, and a choice of *waiting mechanisms* to wait for synchronization delays. Frequently, the right choice depends on run-time factors such as contention and waiting time. As a solution, this thesis investigates reactive synchronization algorithms that dynamically select protocols and waiting mechanisms in response to run-time factors so as to achieve better performance. Through analysis and experimentation, we show that reactive algorithms can achieve close to optimal performance, while incurring minimal run-time overhead.

The first part of this thesis investigates reactive algorithms that dynamically select protocols. We describe a framework for efficiently coordinating concurrent protocol executions with protocol changes, and introduce the notion of *consensus objects* that help preserve correctness. Experiments with reactive algorithms for spin-locks and fetch-and-op demonstrate that the reactive algorithms perform close to the *best* static choice of protocols at any fixed level of contention. With mixed levels of contention, the reactive algorithms can actually *outperform* a static choice of protocols. Measurements of parallel applications show that a bad choice of protocols can result in three times worse performance over the optimal choice. The reactive algorithms are typically within 5% of the best static choice of protocols, and outperform a static choice by 18% in one of the applications.

The second part of this thesis investigates two-phase waiting algorithms for dynamically selecting waiting mechanisms. A two-phase waiting algorithm first polls until the cost of polling reaches a limit L_{poll} before blocking. We prove that under exponentially distributed waiting times, a static choice of L_{poll} results in waiting costs that are at most 1.59 times the cost of an optimal off-line algorithm. Under uniformly distributed waiting times, waiting costs are no more than 1.62 times the cost of an optimal off-line algorithm. These performance bounds are close to the theoretical limit for on-line waiting algorithms. Experimental measurements of several parallel applications demonstrate the robustness of two-phase waiting algorithms, and corroborate the theoretical results.

Thesis Supervisor: Anant Agarwal

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis marks the culmination of six years of graduate study, during which I have had the privilege and pleasure of working with a very capable and highly motivated group of people on the Alewife project. The process of participating in a project to build a complete multiprocessing system is a unique experience that has provided me with a rich source of ideas for my thesis research.

Anant Agarwal, my thesis supervisor and also the Alewife project leader, has been influential in many ways. This thesis would not have been possible without his invaluable guidance and his ability to see the big picture. He is also a great source of motivation, which is especially important for completing a Ph.D. thesis. Thanks also to Steve Ward and Bill Weihl, my thesis readers, for their support, constructive criticisms, and fresh perspectives on the value and limitations of my thesis research.

The Alewife group members provided me with a superb research environment. I enjoyed working with David Kranz on Alewife's system software. John Kubiatowicz, our chief architect, was an endless source of good ideas. Kirk Johnson, Ken Mackenzie, and Donald Yeung were great colleagues and friends for bouncing half-baked research ideas off of. It is also safe to say that none of the Alewife group members would have been able to function without Anne McCarthy to take care of us. Finally, David Chaiken, my office-mate and friend through all these years, was one person I could always turn to, literally, for help. When we entered graduate school together, we decided that we would graduate by the time our office plant reached the ceiling. It has, and we are both graduating.

Outside of the Alewife group, Maurice Herlihy and Nir Shavit were influential in my work on synchronization algorithms. I gained valuable experience from working with them on counting networks in particular, and on multiprocessor coordination in general. My research has also benefited from fruitful discussions with Eric Brewer, Stuart Fiske, Wilson Hsieh, and Debbie Wallach.

Many thanks to all my friends for caring for me and for constantly reminding me that life exists outside of graduate school. I am also indebted to my in-laws for their love and for providing me with a home away from home.

I am eternally grateful for the love, support, and encouragement of my parents and my sister. They have seen me through all my years of education and have shared in all my hardships and successes. I would never have been able to come so far and achieve so much without them. I hope that I can give as much back to them as they have given to me.

Lastly, and most importantly, I wish to acknowledge the unconditional love and support of my wife, Angela (HB) Chang. She has been a great friend and a dependable pillar of strength through all the hard times. She is the main reason that I have been able to keep my sanity during these last few years. This thesis is dedicated to her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

MIMD multiprocessors offer the most flexibility for executing parallel programs by not forcing the hardware to adhere to any particular programming model. However, the asynchrony of the processing nodes requires parallel programs on such machines to synchronize to ensure correctness. For example, threads in a shared-memory parallel program typically execute *synchronization operations*, such as locks and barriers, to enforce mutual exclusion and preserve data dependencies.

The overhead of synchronization can comprise a significant fraction of the execution time of a program. Furthermore, as we run parallel programs on increasingly larger numbers of processors, we can expect the frequency of synchronization operations to increase and comprise an even larger fraction of execution time. Synchronization also tends to serialize computations and limit the achievable speedup of a parallel program. Therefore, it is important to minimize the overhead of synchronization.

There exist several approaches to reducing the impact of synchronization in a parallel program. This thesis focuses on designing efficient algorithms to minimize the overhead of synchronization operations. Multiprocessors typically provide low-level synchronization primitives in hardware, such as atomic *read-modify-write* instructions, and rely on software synchronization algorithms to synthesize higher-level synchronization operations. However, synchronization algorithms that are efficient and robust across a wide range of operating conditions are hard to design because their performance depends on unpredictable run-time factors.

In particular, the designer of a synchronization algorithm has a choice of *protocols* to implement the synchronization operation, and a choice of *waiting mechanisms* to wait for synchronization conditions to be satisfied. Frequently, the best protocol depends on the level of contention, while the best waiting mechanism depends on the length of the waiting

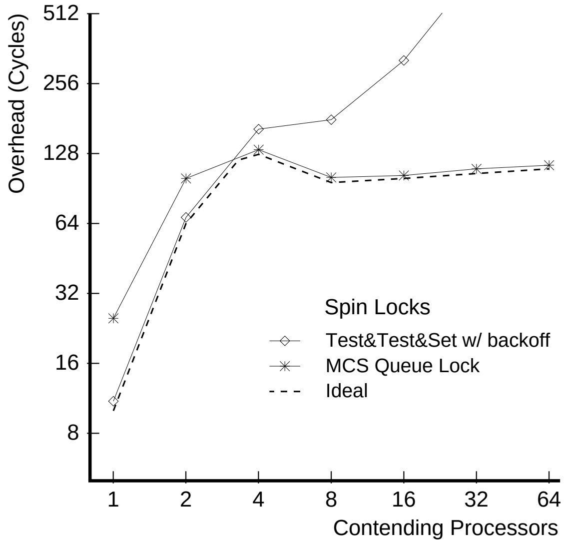


Figure 1.1: *The tradeoff between spin lock algorithms. "Overhead" represents the average number of cycles per completed synchronization operation that is due to the synchronization algorithm in use.*

time. It is difficult to select the best protocol and waiting mechanism without *a priori* knowledge of contention levels and waiting times that will be encountered during program execution.

Choice of Protocols. As an example of the choice of protocols, consider algorithms for mutual-exclusion locks. One algorithm, commonly known as a test-and-set spin lock, uses a protocol that acquires a lock with a *test&set* instruction and releases a lock with a *store* instruction. Although it is a simple and efficient protocol in the absence of contention, its performance degrades drastically under high contention. A remedy is a queuing protocol [43] that constructs a software queue of lock waiters to reduce memory contention. However, queuing comes at the price of a higher latency in the absence of contention. Anderson [5] observes that the choice between the two protocols depends on the level of lock contention.

Figure 1.1 illustrates the tradeoff between the test-and-set spin lock and the MCS queue lock by Mellor-Crummey and Scott [43]. We measured the overhead incurred by these spin lock algorithms on a simulation of the Alewife multiprocessor [4]. Each data point represents the average overhead incurred by the synchronization algorithm for each critical section with P processors contending for the lock. One can view the overhead as the number of cycles the locking algorithm adds to the execution of each critical section. Chapter 3 provides more details of this experimental measurement, but we present the results here to motivate the need for dynamic protocol selection.

The results show that the best protocol to use depends on the level of contention. The MCS queue lock provides the best performance at high contention levels. However, it is twice as expensive as the test-and-set lock when there is no contention due to the extra overhead in maintaining the queue of lock waiters. We would like to design reactive algorithms that dynamically select protocols so that its performance follows the ideal curve. As we will see in Chapter 3, our reactive spin-lock algorithm performs very close to this ideal.

Choice of Waiting Mechanisms. A synchronization algorithm also faces a choice of waiting mechanisms when waiting for synchronization conditions to be satisfied. Two fundamental types of waiting mechanisms are *polling* and *signaling*. With a polling mechanism, the waiting thread periodically polls a synchronization variable and proceeds when the variable attains a desired value. With a signaling mechanism, the waiting thread suspends execution and allows another thread to use the processor. Commonly used waiting mechanisms are spinning and blocking: spinning is a polling mechanism, while blocking is a signaling mechanism. The Alewife multiprocessor provides additional polling and signaling mechanisms through Sparcle, its multithreaded processor [3].

Since a polling mechanism incurs a cost that is proportional to the waiting time, while a signaling mechanism incurs a fixed cost, the choice between a polling and a signaling mechanism depends on the length of the waiting time [47]. Short waiting times favor polling mechanisms while long waiting times favor signaling mechanisms. For example, the cost of blocking a thread on the Alewife multiprocessor is about 500 cycles. Thus, if the waiting time is less than 500 cycles, spinning would be more efficient than blocking.

1.1 Reactive Synchronization Algorithms

The preceding discussion illustrates the difficulty of designing efficient synchronization algorithms. The best choice depends on run-time factors that are hard to predict. Furthermore, a bad choice may result in unnecessarily high synchronization overheads. Given the difficulty of making the right choice of protocols and waiting mechanisms, current practice is to rely on the programmer to make the choice. However, this places an unnecessarily heavy burden on the programmer, especially since the choice is run-time dependent.

When an optimal static choice of protocols and waiting mechanisms cannot be made,

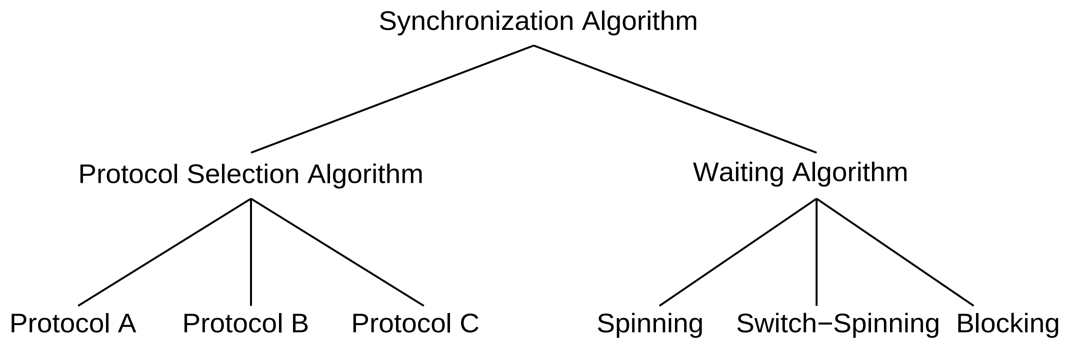


Figure 1.2: *The components of a reactive synchronization algorithm.*

the obvious alternative is to turn to run-time techniques for making the choice dynamically. This thesis addresses the question: Is it possible to select protocols and waiting mechanisms dynamically and achieve close to optimal performance? The results of this thesis show that this goal is indeed possible.

This thesis designs, implements and analyzes *reactive synchronization algorithms* that automatically choose the best protocols and waiting mechanisms to use. Reactive synchronization algorithms achieve this by monitoring and responding to run-time conditions. Figure 1.2 illustrates the components of a reactive synchronization algorithm. It is composed of a protocol selection algorithm and a waiting algorithm. The protocol selection algorithm is responsible for choosing the best protocol to implement the synchronization operation, while the waiting algorithm is responsible for choosing the best waiting mechanism to wait for synchronization delays.

The main challenge in designing reactive synchronization algorithms is ensuring that the run-time overhead of making the choices be kept to a minimum. Not only is it essential that the run-time selection be correct, but it must also be performed with minimal overhead and yield improved performance. This places a limit on the complexity of the algorithms for detecting run-time conditions and deciding which protocol and waiting mechanism to use.

Since dynamic protocol and waiting mechanism selection are instances of on-line problems, we rely on previous research on competitive algorithms [9, 41] to help design the reactive algorithms. An *on-line problem* is one in which an algorithm must process a sequence of requests without knowledge of future requests. Previous theoretical research has designed competitive algorithms for solving on-line problems with performance that is at most a constant factor worse than the performance of an optimal off-line algorithm. This constant is termed the *competitive factor*.

To demonstrate the performance benefits of reactive algorithms, this thesis implements protocol selection algorithms and waiting algorithms for several common synchronization operations. It evaluates their performance against the best known synchronization algorithms. Experimental results demonstrate that the reactive synchronization algorithms yield robust performance over a wide range of operating conditions while incurring minimal overhead. In most cases, the reactive algorithms perform close to or better than the best static choice of protocols and waiting mechanisms.

In designing and evaluating the performance of reactive algorithms, we were careful to avoid relying on any specific features of the Alewife multiprocessor. The reactive algorithms rely only on conventional shared-memory primitives, and the results of this thesis should be applicable to most multiprocessor architectures that support the shared-memory abstraction.

An important benefit of reactive synchronization algorithms is that they relieve the programmer from the difficult, if not impossible, task of predicting run-time conditions to minimize synchronization costs. The reactive synchronization algorithms can be provided as a library of synchronization operations that a programmer can link with his program. Although the protocol and waiting mechanism in use may change dynamically, the interface to the application program remains constant. Thus, the process of selecting the best protocols and waiting mechanisms is entirely the responsibility of the reactive synchronization algorithm, and is completely invisible to the programmer using the synchronization library.

We subdivide this thesis into two major parts, each corresponding to one of the components of a reactive synchronization algorithm. The first part is concerned with protocol selection, while the second part is concerned with waiting mechanism selection. Dynamic protocol selection and waiting mechanism selection involve different issues and problems. Let us consider the issues involved in each of these parts in more detail.

1.1.1 Protocol Selection

There exists a multitude of protocols for common synchronization operations such as locks, barriers, and fetch-and-op. Recent research on synchronization algorithms has resulted in protocols that are optimized for performance under high contention. Unfortunately, under low contention, these so-called "scalable" protocols come at the price of higher overheads than simpler protocols

This tradeoff makes the best choice of protocols depend on run-time levels of contention. To further complicate the choice, each of many synchronization objects in a program may experience a different level of contention. For example, when traversing a directory data structure that is organized as a tree, locks at the root of the tree are likely to be more heavily contended than locks that are close to the leaves of the tree. Contention levels at each object may also be data-dependent and may vary over time. Given the complexity of selecting the best protocol for each synchronization object, current practice is simply to use the same protocol.

Our approach is to design run-time algorithms to select protocols dynamically, based on the level of contention. The goal is to use the best protocol for each synchronization object even with time-varying contention levels. Although the idea of dynamically selecting protocols is intuitively appealing, there has not been any experimental research on its feasibility and performance benefits.

There are two main challenges to selecting protocols dynamically. First is the challenge of designing efficient *methods* for selecting and changing protocols. That is, *how* do we select and change protocols efficiently? Multiple processes may be trying to execute the synchronization operation at the same time, and keeping them in constant agreement on the protocol to use may be as hard as implementing the synchronization operation itself. Before this thesis research, it was not clear if the overhead of managing the access of multiple processes to multiple protocols would be prohibitively expensive. We provide a framework for reasoning about dynamic protocol selection and introduce the notion of *consensus objects* that our reactive algorithms use to help ensure correct operation in the face of dynamic protocol changes.

Second is the challenge of designing an intelligent *policy* for changing protocols. That is, *when* should we change protocols? A reactive algorithm that finds itself using a sub-optimal protocol needs to decide if it should switch to a better protocol. Because switching from one protocol to another incurs a significant fixed cost, a naive policy that switches protocols immediately may thrash between protocols and perform badly. The decision to switch protocols depends on the future behavior of contention levels. This is an instance of an on-line problem, and we present a 3-competitive algorithm for deciding when to change protocols.

We present empirical results that demonstrate that under fixed contention levels, the reactive algorithms performs close to the *best* static choice of protocols at any level of contention. Furthermore, with mixed levels of contention, either across multiple synchronization objects or over time, the reactive algorithms *outperform* conventional algorithms with fixed protocols, unless extremely frequent protocol changes are required.

Measurements of the running times of several parallel applications show that the bad choice of protocols can result in three times worse performance over the optimal choice. The application running times with the reactive algorithms are typically within 5% of the performance of the best static choice. In one of the applications, the reactive algorithm outperformed a static choice of protocols by 18%.

1.1.2 Waiting Mechanism Selection

Waiting is a fundamental part of synchronization, regardless of the protocol being used to implement the synchronization operation. While waiting for synchronization delays, a thread has a choice of polling or signaling waiting mechanisms to use. Multiprocessors traditionally provide spinning and blocking as waiting mechanisms, and rely on the programmer to make the right choice. Spinning, a polling mechanism, consumes processor cycles that could be used for executing other threads. Blocking, a signaling mechanism, incurs a significant fixed cost because of the need to save and restore processor state.

Since a polling mechanism incurs a cost that is proportional to the waiting time, while a signaling mechanism incurs a fixed cost, short waiting times favor polling mechanisms while long waiting times favor signaling mechanisms. However, it is hard to make a correct choice without *a priori* knowledge of wait times, and run-time techniques are needed to select the appropriate waiting mechanism.

Unlike switching among protocols, switching among waiting mechanisms is a local operation that does not need to be coordinated among participating processes. It is easy to provide a mechanism for dynamically choosing waiting mechanisms, and existing multiprocessor systems provide the option of spinning vs. blocking.

The main challenge to dynamically selecting waiting mechanisms is in designing an intelligent *policy* for deciding *when* to switch from a polling mechanism to a signaling mechanism. Since this is another instance of an on-line problem, competitive techniques can be used to bound the worst case cost of a waiting algorithm.

A popular algorithm for selecting waiting mechanisms is the *two-phase waiting algorithm* [47], where a waiting thread first polls until the cost of polling reaches a limit L_{poll}. If further waiting is necessary, the thread resorts to a signaling mechanism and incurs a fixed cost. The choice of L_{poll} is key to the performance of a two-phase waiting algorithm.

With appropriate choices of L_{poll}, we can prove that the cost of two-phase waiting is not more than a small constant factor more than the cost of an optimal off-line algorithm. For example, setting L_{poll} equal to the cost of blocking a thread yields a 2-competitive waiting algorithm. Karlin *et al.* [26] present a randomized algorithm for selecting L_{poll} that achieves a competitive factor of 1.58. They also prove a lower bound of 1.58 on the competitive factor of any on-line waiting algorithm.

This thesis investigates two-phase waiting algorithms in the context of a multiprocessing system with lightweight threads. In such a system, the cost of blocking is small enough that the run-time overhead of determining L_{poll} must be minimized. We show how to determine the value of L_{poll} statically, and still achieve close to the optimal on-line competitive factor of 1.58. We also measure waiting times and demonstrate the robustness of two-phase waiting algorithms in a number of parallel applications.

The choice of waiting mechanisms has a significant effect on the running time of the applications that we studied. A bad choice of waiting mechanisms can result in 2.4 times worse performance than the optimal choice. However, the two-phase waiting algorithm is typically within 6.6% of the best static choice of waiting mechanisms.

1.2 Contributions of this Thesis

This thesis is both a theoretical and empirical study of using run-time adaptivity to reduce the cost of synchronization. It demonstrates the performance benefits of tailoring the protocol and waiting mechanism to run-time conditions. In performing the study, this thesis makes the following contributions.

- It introduces and evaluates the idea of dynamically choosing synchronization protocols in response to run-time conditions. Previous adaptive approaches to synchronization have considered altering the waiting or scheduling policy of a protocol, but have not considered actually changing the protocol in use.

- It presents a framework for designing and reasoning about algorithms that select and change protocols dynamically. Dynamic protocol selection presents a coordination problem that has to be solved efficiently. We introduce the notion of *consensus objects* that allows protocol selection to be implemented efficiently. With this method, the overhead of protocol selection is only as small as a few conditional branches in the common case.

- It presents reactive algorithms for spin locks and fetch-and-op. The reactive algorithms are experimentally shown to be more robust than the best existing algorithms for spin locks and fetch-and-op. Section 3.7 and Appendix C overview the implementation process and present pseudo-code listings of these algorithms.

- It significantly extends previous work on waiting algorithms, both analytically and experimentally, by considering practical aspects of a scalable, parallel machine environment. It introduces the notion of *restricted adversaries*, which is a more realistic and useful model than traditionally assumed adversaries.

- It proves that under restricted adversaries, static choices of L_{poll} for two-phase waiting can yield close to optimal on-line competitive factors. Under exponentially distributed wait times, setting L_{poll} to 0.54 times the cost of blocking yields a 1.58-competitive waiting algorithm, while under uniformly distributed waiting times, setting L_{poll} to 0.62 times the cost of blocking yields a 1.62-competitive waiting algorithm.

- It evaluates the performance of two-phase waiting in applications running on a scalable multiprocessor architecture with a highly optimized run-time system. The analysis considers a variety of synchronization types. It presents experimental results on waiting times and execution times of several applications under different waiting algorithms.

1.3 Terminology

This thesis uses the following terminology to describe synchronization algorithms and the sources of synchronization overhead.

Protocol – A synchronization protocol implements a synchronization operation. For example, the MCS queue lock protocol [43] implements a mutual exclusion lock and a combining-tree protocol [57] implements a barrier.

Waiting mechanism – A waiting mechanism waits for synchronization conditions to be satisfied. Spinning and blocking are two common waiting mechanisms. A multithreaded processor may provide alternative waiting mechanisms, such as switch-spinning and switch-blocking [39, 16].

Waiting time – Synchronization waiting time is the interval from when a thread begins waiting on a synchronization condition to when the synchronization condition is satisfied and the waiting thread is allowed to proceed.

Synchronization cost – We define synchronization cost as the number of processor cycles consumed while performing synchronization operations. When synchronizing, a processor is either busy executing the synchronization protocol, or waiting for some synchronization condition to be satisfied. Thus, the cost of synchronization can be subdivided into the following two costs.

Protocol cost – The number of cycles spent executing the synchronization protocol in order to perform a synchronization operation. This cost represents actual computation that cannot be avoided when performing a synchronization operation.

Waiting cost – The cost of waiting for a synchronization condition to be satisfied during a synchronization operation. This cost depends on the waiting time and on the waiting mechanisms that are used. Since no useful computation is performed while waiting, this cost can be reduced by switching processor execution to another thread.

1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 provides relevant background material on the theoretical analysis and the experimental platform used for designing and evaluating the synchronization algorithms. Chapter 3 describes protocol selection algorithms. It provides a framework for reasoning about and designing protocol selection algorithms. It also presents and evaluates reactive algorithms for spin locks and fetch-and-op. Chapter 4 describes waiting algorithms. It presents theoretical and empirical analyses of two-phase waiting algorithms. Chapter 5 reviews related research in designing efficient synchronization algorithms, and reviews complementary approaches to reducing the cost of synchronization. Chapter 6 summarizes the thesis and presents suggestions for future work. Appendix A provides a brief description of the Spec language and Appendix B uses Spec to provide precise descriptions of the framework for implementing protocol selection algorithms. Finally, Appendix C presents pseudo-code listings of the reactive fetch-and-op algorithm.

Chapter 2

Background

This chapter provides background material on the theoretical analysis and the experimental platform and methodology used in this thesis. It first overviews previous research on competitive on-line algorithms and shows how they relate to the design of reactive synchronization algorithms. It then describes the multiprocessing platform used for experimental evaluation of the reactive algorithms. Specifically, we describe relevant features of the Alewife multiprocessor [4] and the organization of the Alewife simulator on which we ran most of the experiments.

2.1 Competitive On-Line Algorithms

An *on-line problem* is one in which an algorithm must process a sequence of requests without knowledge of future requests. Usually, each request can be satisfied in more than one way, and an *on-line algorithm* must choose how to satisfy a request so as to minimize the total cost of satisfying a sequence of requests. The difficulty is that the on-line algorithm has to make a decision based only on knowledge of the current and past requests, although the decision may affect the cost of satisfying future requests.

The amortized analysis of on-line algorithms has been extensively studied in the theory community in recent years [9, 41, 26]. The objective has been to design on-line algorithms that are within a small constant factor of the performance of an optimal off-line algorithm that has complete knowledge of future requests. Karlin *et al.* [27] coined the term *c-competitive* to describe such algorithms. A c-competitive algorithm has a cost that is at most c times the cost of an optimal off-line algorithm plus a fixed constant term, for any sequence of requests. c is termed the *competitive factor*.

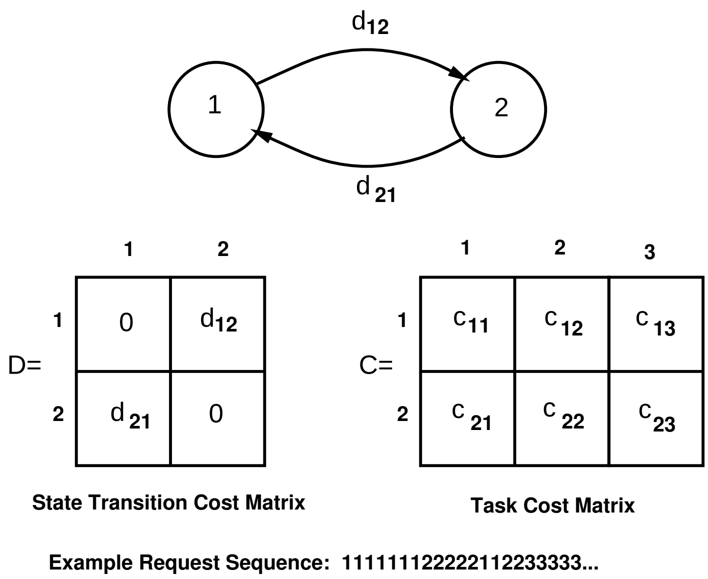


Figure 2.1: *An example task system with 2 states and 3 tasks.*

2.1.1 Task Systems

Borodin, Linial and Saks [9] formalized the definition of an on-line problem and called it a *task system*. They also designed a $2n-1$-competitive algorithm for task systems, where n is the number of states in the task system. To paraphrase the definition of a task system from [41],

> A task system consists of a set of n states, a set of m tasks, an n by n state transition cost matrix D, where d_{ij} is the cost of changing from state i to state j, and an n by m task cost matrix C, where c_{ij} is the cost of processing task j in state i.
>
> A sequence of requests $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(N)$ is to be processed by the system. Each request, $\sigma(i)$, is one of the tasks. An algorithm for a task system chooses which state to use to satisfy each of the requests.

Figure 2.1 illustrates a task system with 2 states and 3 tasks. The cost incurred by an on-line algorithm is the sum total of the costs of processing the tasks and the costs of the state transitions. An on-line algorithm attempts to prescribe a schedule of state transitions such that the cost of the on-line algorithm is minimized. We assume *lookahead-one* task systems, where the algorithm is allowed to change states *before* servicing the current request.

A protocol selection algorithm can be thought of as an on-line algorithm for processing a sequence of synchronization requests. It must choose which of several protocols to use to satisfy each synchronization request. We can easily map the problem of protocol selection onto a task system. Each state of a task system represents a protocol and each task represents a synchronization request under a particular run-time condition. The state transition cost matrix represents the costs of switching protocols, and the task cost matrix represents the cost of satisfying a synchronization request with a given protocol under different run-time conditions.

A waiting algorithm can also be thought of as an on-line algorithm for processing a sequence of wait requests. It must choose which of several waiting mechanisms to satisfy each request. Again, we can map the problem of waiting mechanism selection onto a task system. Each state of a task system represents a waiting mechanism and each task represents a request to wait. The state transition cost matrix represents the costs of switching from a polling to a signaling waiting mechanism, and the task cost matrix represents the cost of waiting under different waiting mechanisms.

This observation allows us to use the results of previous research on competitive on-line algorithms to help design competitive protocol selection algorithms and waiting algorithms. In Chapters 3 and 4, we will construct task systems that represent the on-line problem of choosing among protocols and waiting mechanisms. In Chapter 3, we use an on-line algorithm by Borodin, Linial and Saks to design a 3-competitive algorithm for deciding when to change protocols. In Chapter 4, based on work by Karlin *et al.* [26], we will use probabilistic analysis to design a 1.58-competitive algorithm for deciding between polling and signaling waiting mechanisms.

2.2 Experimental Platform

To investigate the performance characteristics of reactive synchronization algorithms, we run a set of synthetic and application benchmarks that exercise the synchronization algorithms. The primary goals of the experiments are to corroborate the theoretical analysis and to measure the performance characteristics of the algorithms. We compare the performance of the reactive algorithms with the best existing algorithms, both in synthetic and application benchmarks. These experiments were run on a simulation of the Alewife multiprocessor. The Alewife multiprocessor is representative of a scalable shared-memory architecture based on distributed nodes that communicate via an interconnection network.

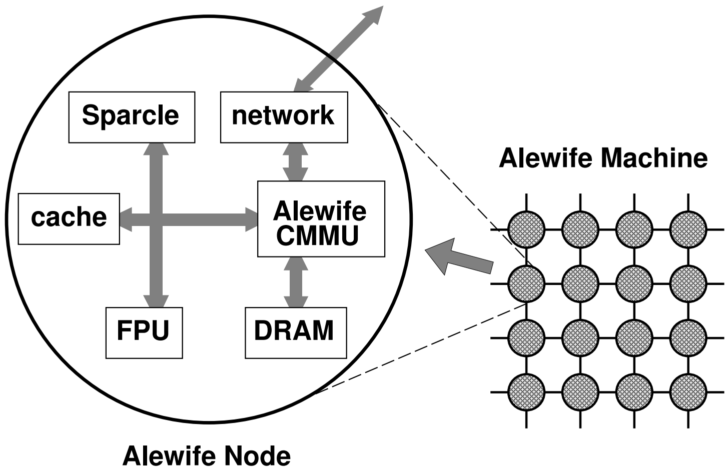


Figure 2.2: *An Alewife node.*

2.2.1 The Alewife Multiprocessor

The MIT Alewife multiprocessor [4] is a cache-coherent, distributed-memory multiprocessor that supports the shared-memory programming abstraction. Figure 2.2 illustrates the high-level organization of an Alewife node. Each node consists of a Sparcle processor [3], an FPU, 64KB of cache memory, a 4MB portion of globally-addressable memory, the Caltech MRC network router, and the Alewife Communications and Memory Management Unit (CMMU) [32]. The current prototype is designed to run at 33MHz.

Sparcle is a modified SPARC processor that supports multithreading. Multiple threads can be resident on the processor at once, and the processor can context switch from one processor-resident thread to another in 14 cycles. It allows us to consider multithreading as providing additional waiting mechanisms in our research on waiting algorithms.

The CMMU implements a cache-coherent globally-shared address space with the LimitLESS cache-coherence protocol [12]. The LimitLESS cache-coherence protocol maintains a small, fixed number of directory pointers in hardware, and relies on software trap handlers to handle cache-coherence actions when the number of read copies of a cache block exceeds the limited number of hardware directory pointers. The current implementation of the Alewife CMMU has 5 hardware directory pointers per cache line.

The CMMU also interfaces the Sparcle processor to the interconnection network, allowing the use of an efficient message-passing interface for communication [31]. The LimitLESS protocol relies on this interface to handle coherence operations in software. The message interface also allows us to use message-passing operations to implement synchronization operations.

2.2.2 NWO: The Alewife Simulator

At the time that this thesis research was performed, the Alewife CMMU was in the process of being fabricated. Since the actual hardware was not available then, most of the experimental data presented in this thesis were gathered from an accurate cycle-by-cycle simulation of the machine. The Alewife simulator, dubbed NWO, has been used extensively for validating the hardware design and for development of system software and applications. NWO is binary-compatible with the Alewife machine: object code generated by the compiler can be run unmodified on either the simulator or on the actual machine. NWO is faithful enough to the hardware design that it exposed many Alewife hardware errors during the design phase. It also allowed us to implement Alewife's run-time system even before hardware was available.

Figure 2.3 illustrates the organization of the simulator, which is coupled with a programming system that supports C and Mul-T (a dialect of Lisp). All code is compiled with the ORBIT [29] compiler, an optimizing compiler for Scheme. The ORBIT compiler was extended for generating parallel code.

The primary drawback of NWO is its slow simulation speed: it provides accuracy at the price of simulation speed. On a SPARCstation 10, it simulates approximately 2000 processor cycles per second. Fortunately, a parallel version of NWO runs on a Thinking Machines Corporation CM-5 [37], allowing us to simulate a large number of processing nodes in a reasonable amount of time.

A 16-node Alewife prototype recently became operational in June, 1994. We will present data from the real machine that validates some of the results gathered from the simulations. The prototype currently runs reliably at a clock speed of 20MHz.

2.2.3 Synchronization Support

The Alewife multiprocessor was designed to provide a variety of mechanisms to help implement synchronization operations efficiently.

- Alewife provides two basic hardware primitives for synchronization: an atomic *fetch&store* instruction, and a set of instructions that manipulate *full-empty bits*[53, 28] associated with each memory word. These instructions are directly supported by the Alewife CMMU, and are cache-coherent.

- Alewife supports automatic detection of unresolved *future*s [22] by using the least significant bit of a data word as a tag bit, and by trapping on misaligned addresses in

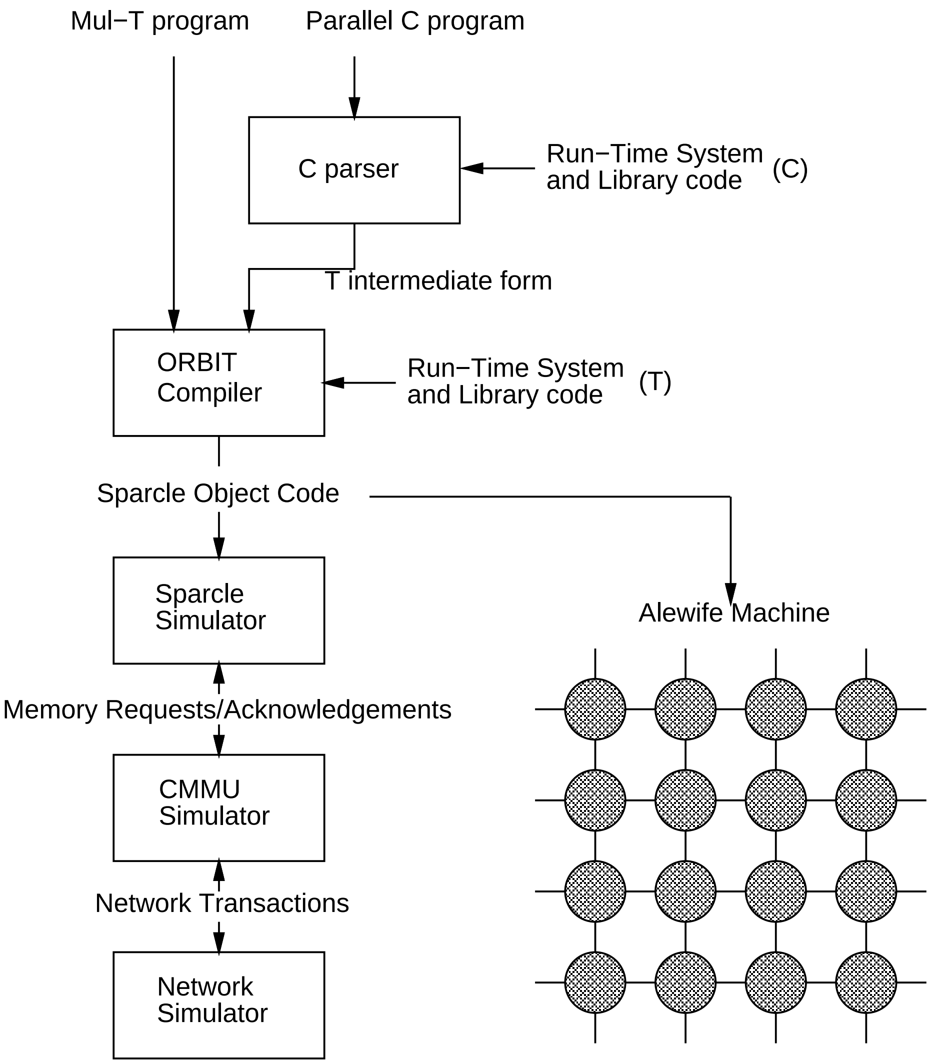


Figure 2.3: *Organization of the NWO simulation system for Alewife.*

memory access instructions.

- Alewife allows software to directly access the underlying message layer [31], providing the opportunity for software to use messages to implement shared-memory synchronization operations.
- Lastly, Alewife's processor implements a coarse-grained version of multithreading, called block multithreading [33], that can be used to lessen the cost of waiting for synchronization.

Except for *fetch&store* to serve as an atomic read-modify-write primitive, the contributions and conclusions of this research do not depend on the availability of these synchronization features of the Alewife multiprocessor. We were careful to avoid using any esoteric features of the Alewife architecture so that the results of this thesis are applicable to other multiprocessor architectures.

Nevertheless, we do investigate the implications of having such support. In research on protocol selection algorithms, we study the benefits of using message passing to implement synchronization protocols and the tradeoffs that arise. In research on waiting algorithms, we consider additional waiting mechanisms that are made possible through multithreading. We also consider the performance of waiting algorithms for several synchronization types, such as *future*s [22] and I-structures [6], that are implemented with tagging and full-empty bits.

2.2.4 Run-Time System Assumptions

The scheduling policy and the run-time overhead of thread management have a significant impact on the performance of waiting algorithms. In Alewife's run-time system, thread scheduling is non-preemptive so that spin-waiting indefinitely may starve processes that have been swapped out. The run-time system implements a very streamlined and minimal thread management system, such that the cost of loading and unloading a thread is very small. The cost of blocking a thread in the current implementation is less than 500 cycles. This places a limit on the complexity of a run-time algorithm to decide between waiting mechanisms.

Chapter 3

Protocol Selection Algorithms

This chapter focuses on the design and analysis of protocol selection algorithms. Protocol selection algorithms promise the potential for reducing the cost of synchronization by dynamically selecting the best protocol to use in response to the run-time conditions experienced by each synchronization operation. However, in order to realize this potential, we have to overcome two hurdles.

First, we have to ensure that the run-time cost of coordinating concurrent access to multiple protocols in the face of dynamic protocol changes does not overwhelm the benefits of using the best protocol. Second, we have to make intelligent on-line decisions of when to change protocols. This chapter will describe our solutions to each of these problems, and demonstrate the benefits of dynamically selecting protocols.

As an example of the potential benefits of dynamic protocol selection, we first review several protocols for spin locks and fetch-and-op, and show how the most efficient protocol depends on the level of contention experienced at run time. We then describe a framework for designing efficient protocol selection algorithms. Using the design framework, we implement reactive spin lock and fetch-and-op algorithms that choose among several well-known shared-memory and message-passing protocols.

Experimental measurements demonstrate that the reactive algorithms perform close to the *best* static choice of protocols at all levels of contention. Furthermore, with mixed levels of contention, the reactive algorithms outperform passive algorithms with fixed protocols, provided that contention levels do not change too frequently. Measurements of the running times of several parallel applications show that a bad choice of protocols can result in three times worse performance over the optimal choice. The application running times with the reactive algorithms are typically within 5% of the performance of the *best* static choice, and are at worst 18% longer. In one of the applications, the reactive algorithm outperformed the best static choice by 18%.

3.1 Motivation

To motivate the need for dynamic protocol selection, we first review existing algorithms for spin locks and fetch-and-op. These algorithms are *passive* in the sense that they use a fixed protocol regardless of the run-time level of contention. This review provides a description of the performance characteristics of the passive algorithms and of the tradeoffs among them. Performance measurements of these passive algorithms on the Alewife multiprocessor demonstrate how the best protocol depends on the level of contention.

3.1.1 Passive Spin-Lock Algorithms

The test-and-set Algorithm This algorithm uses a very simple protocol. A process requests a lock by repeatedly executing a *test&set* instruction on a boolean flag until it successfully changes the flag from false to true. A process releases a lock by setting the flag to false.

```
procedure lock(l)                          procedure unlock(l)
    repeat while test_and_set(l)               l^ := false
```

The primary problem with a test-and-set protocol is that its performance degrades drastically in the presence of contention: waiting processes continuously poll the lock, resulting in an overwhelming amount of bus or network traffic.

The test-and-test-and-set Algorithm Segall and Rudolph [50] proposed the test-and-test-and-set algorithm for reducing bus or network traffic on cache-coherent machines. It uses a protocol that waits by read-polling the lock:

```
procedure lock(l)
    repeat while (l^ or test_and_set(l))
```

On a cache-coherent machine, the lock variable will be read-cached, thus avoiding communication while the lock is held. However a significant amount of communication traffic is still generated when the lock is released due to the ensuing cache invalidations and updates. With small critical sections, this transient behavior dominates and read-polling can generate as much communication traffic as polling with *test&set* [5].

Recent research has resulted in more sophisticated protocols that alleviate the detrimental effects of contention [5, 19, 43]. The research demonstrates that test-and-set with randomized exponential backoff and queuing are the most promising spin-lock protocols.

Exponential Backoff Anderson [5] proposed exponential backoff as a way of reducing contention for spin locks. Agarwal and Cherian [2] independently proposed exponential backoff for reducing contention at barriers. The idea is to have each waiting process introduce some delay between lock accesses in the test-and-set or test-and-test-and-set algorithms. This reduces the amount of unsuccessful *test&set* attempts and avoids excessive communication traffic under high contention.

```
procedure lock(l)
    delay : integer := INITIAL_DELAY
    repeat while test_and_set(l)
        pause(random(delay))
        delay := MAX(delay*2, MAX_DELAY)
```

Anderson found that the best performance is achieved with randomized exponential backoff, where the mean delay is doubled after each failed attempt and halved after each successful attempt. There also needs to be a maximum bound on the mean delay, proportional to the level of lock contention. Otherwise, a waiting processor may back off to an arbitrarily large delay, leading to poor response time when the lock becomes free. The maximum bound should be large enough to accommodate the *maximum* possible number of contending processors.

Randomized exponential backoff can be thought of as providing "probabilistic" queuing of lock waiters: through adaptivity and randomization, it attempts to spread out waiting processes in time. Henceforth, we will refer to test-and-set with exponential backoff simply as test-and-set locks, and test-and-test-and-set with exponential backoff simply as test-and-test-and-set locks.

Queue Locks Queue locks explicitly construct a queue of waiters in software. Memory contention is reduced because each waiter spins on a different memory location and only one lock waiter is signaled when the lock is released. Queue locks have the additional advantage of providing fair access to the lock. Several queue lock protocols were developed independently by Anderson [5], Graunke and Thakkar [19], and Mellor-Crummey and Scott [43].

All three protocols scale well with the level of contention. However, the first two queue locks require space per lock proportional to the number of processors, and Anderson's queue lock has a high single-processor latency on machines that do not support atomic *fetch&increment* directly in hardware. In this thesis, we use the Mellor-Crummey and Scott (MCS) protocol for queue locks because it has the best performance among the queuing protocols on our system.

```
type qnode = record
    next   : ^qnode
    locked : boolean

type lock = ^qnode

procedure lock(L:^lock, I:^qnode)
    I->next = nil;
    pred:^qnode := fetch_and_store(L, I)
    if pred != nil
        I->locked := true
        pred->next := I
        repeat while I->locked

procedure unlock(L :^lock, I : ^qnode)
    if I->next = nil
        if compare_and_swap(L, I, nil)
            return
        repeat while I->next = nil
    I->next->locked := false
```

Figure 3.1: *The MCS queue lock protocol by Mellor-Crummey and Scott.*

The MCS queue lock protocol appears in Figure 3.1. The protocol maintains a pointer to the tail of a queue of lock waiters. The lock is free if it points to an empty queue, and is busy otherwise. The process at the head of the queue owns the lock, and each process on the queue has a pointer to its successor. To acquire a lock, a process appends itself to the tail of the queue with a *fetch&store* instruction. If the queue was empty, the process owns the lock; otherwise it waits for a signal from its predecessor. To release a lock, a process checks to see if it has a waiting successor. If so, it signals that successor, otherwise it empties the queue with a *compare&swap* instruction. An alternative version empties the queue with a *fetch&store* instruction, but requires more complicated code to handle a race condition. See [43] for a more complete description of the MCS lock. In our experiments, we use the version of the MCS lock that does not rely on compare-and-swap since Alewife does not have a *compare&swap* instruction.

3.1.2 Passive Fetch-and-Op Algorithms

Fetch-and-op is a useful primitive for implementing higher-level synchronization operations. When the operation is combinable [30], *e.g.*, in fetch-and-add, combining techniques can be used to compute the operation in parallel. Fetch-and-op was deemed important enough for the designers of the NYU Ultracomputer [17] to include hardware support in its interconnection network for combining fetch-and-op requests to the same memory location. The Stanford DASH multiprocessor [38] supports fetch-and-increment and fetch-and-decrement directly in its cache coherence protocol, although without combining.

In the absence of special hardware support, several software algorithms can be used to implement fetch-and-op. We consider the following in this paper.

Lock-Based Fetch-and-Op A straightforward implementation of fetch-and-op is to protect access to the fetch-and-op variable with a mutual exclusion lock. In particular, either the test-and-set lock or the queue lock described above can be used here. To execute a fetch-and-op, a process acquires the lock, updates the value of the fetch-and-op variable, and releases the lock.

Software Combining Tree A drawback of a centralized, lock-based implementation of fetch-and-op is that it may unnecessarily serialize fetch-and-op operations. Software combining protocols [57] can be used to compute the fetch-and-op in parallel. The idea is to combine multiple operations from different processes into a single operation whenever possible.

The fetch-and-op variable is stored in the root of a software combining tree, and combining takes place at the internal nodes of the tree. If two processes arrive simultaneously at a node in the tree, their operations are combined. One of the processes proceeds up the tree with the combined operation while the other waits at that node. When a process reaches the root of the combining tree, it updates the value of the fetch-and-op variable and proceeds down the combining tree, distributing results to processes that it combined with while ascending the tree.

In this thesis, we use the software combining tree algorithm for fetch-and-op presented by Goodman, Vernon and Woest in [15]. We present the pseudo-code of the algorithm in Appendix C since it is too long to reproduce here.

3.1.3 The Problem with Passive Algorithms

The problem with a passive algorithm is that it fixes its choice of protocols, and is thus optimized for a certain level of contention/concurrency at each synchronization operation. We measured the performance of the above spin lock and fetch-and-op algorithms by having each processor loop continuously, performing a synchronization operation on the same synchronization object at each iteration.

We compare the performance of the algorithms by measuring the average elapsed time in between successive synchronization operations at that object. Part of this time is due to

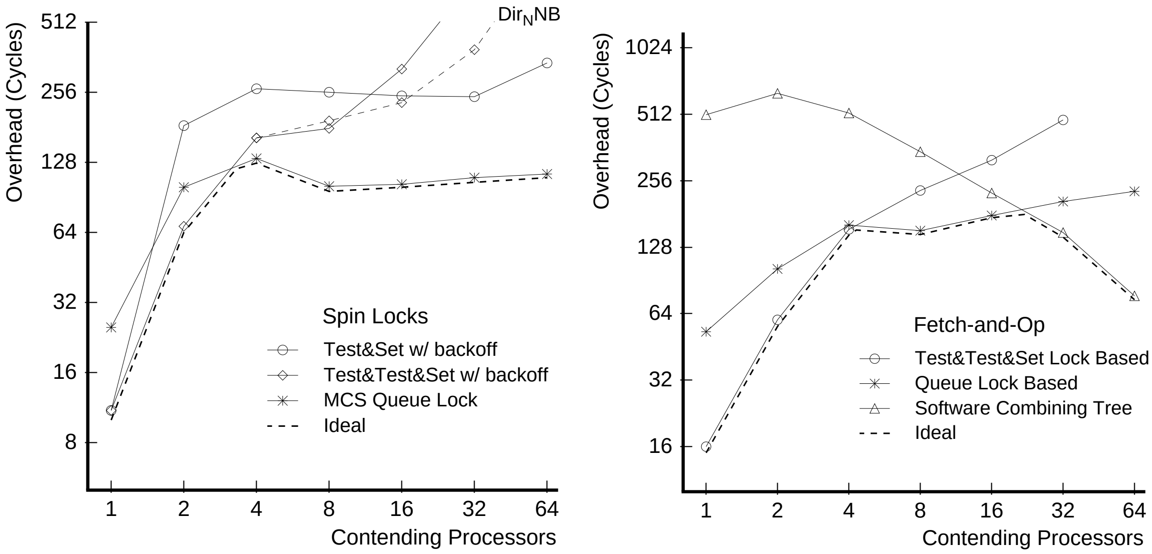


Figure 3.2: *Baseline performance of passive spin lock and fetch-and-op algorithms. "Overhead" represents the average number of cycles per completed synchronization operation that is due to the synchronization algorithm in use.*

the latency introduced by the test loop itself. We filter out the test loop overhead from our measurements by subtracting the time the test would have taken, given zero-overhead synchronization operations, from the actual measured time. The resulting overhead represents the average number of cycles per synchronization operation that is due to the synchronization algorithm in use. Section 3.5.1 provides more details on the experiment and on how the measurements are derived.

Figure 3.2 presents the results of running this experiment on the Alewife simulator. For spin locks, each data point represents the average number of cycles due to the spin lock algorithm for each critical section with P processors contending for the lock. For fetch-and-op, each data point represents the average number of cycles due to the fetch-and-op algorithm for each fetch-and-op with P processors attempting to perform the operation.

Spin Locks The spin-lock results show that the MCS queue lock provides the best performance at high contention levels. However, it is twice as expensive as the test-and-set lock when there is no contention. This is due to the protocol cost of maintaining the queue of lock waiters. The results indicate that no single protocol has the best performance across contention levels. These measurements are consistent with previously reported results in [5, 43, 19]. It is clear that a reactive spin lock algorithm should select the test-and-test-and-set protocol when contention is low and the queue protocol when contention is high.

The test-and-test-and-set protocol has the lowest overhead at low contention levels, and outperforms the test-and-set protocol. This is due to the interaction of exponential backoff and the two protocols. Recall that with exponential backoff, the backoff interval is doubled after each failed *test&set* attempt. Since the test-and-set protocol polls the lock with *test&set*, while the test-and-test-and-set protocol polls the lock with reads, the test-and-set protocol experiences more failed *test&set* attempts. This has the effect of making the test-and-set protocol back off to a larger delay than the test-and-test-and-set protocol. Since the backoff limit is set to accommodate the *maximum* possible number of lock requesters, lock waiters back off too far under test-and-set at low contention levels, leading to poor response times when the lock is released. In the experiments, the backoff limit was set to accommodate 64 contending processors.

However, the test-and-test-and-set protocol does not scale as well as the test-and-set protocol. This is because Alewife uses a directory-based cache coherence protocol that issues cache invalidations sequentially. The test-and-test-and-set protocol has the effect of delaying a lock release as read-cached copies of the lock are invalidated sequentially. In fact, this protocol would scale poorly on any cache-coherent architecture without hardware-supported broadcast for similar reasons.

Another reason for the poor scalability of the test-and-test-and-set protocol in Alewife is the limited number of hardware directory pointers for keeping track of cached copies. When the worker set of a cache line exceeds the number of hardware pointers, a trap handler is invoked to extend the directory in software. To investigate the performance penalty of a limited number of hardware pointers, we also simulated test-and-test-and-set on a full-map directory architecture which handles all coherence actions in hardware. We plot the results as the curve labeled $Dir_N NB$. The results show that while the full-map directory reduces the overhead of the test-and-test-and-set protocol at high contention levels, it still does not scale well.

Fetch-and-Op The fetch-and-op results show that software combining succeeds in parallelizing the overhead of fetch-and-op: as contention (and hence parallelism) increases, the overhead drops. This happens because the overhead of traversing a tree is amortized among the processes participating in the combining tree. In contrast, the overhead increases with contention for the lock-based algorithms.

However, when contention is low, software combining incurs an unnecessarily large overhead from requiring a process to traverse the combining tree, even when there are no other processes to combine fetch-and-op operations with. Here, the lock-based algorithms perform better because they have a much smaller protocol cost. The tradeoff between the two lock-based fetch-and-op algorithms is similar to the tradeoff between the test-and-test-and-set lock and queue lock protocols, as can be expected from the results on spin locks.

Thus, we have a contention-dependent choice of protocols for fetch-and-op. When contention is low, we should use the lock-based protocols for minimal latency. When contention is high, we should use the combining tree protocol for higher throughput. In fact the right choice of protocols is even more important for fetch-and-op than for spin locks. The performance difference between the best and worst protocols spans several orders of magnitude.

In summary, these results demonstrate how the best choice of protocols depends on the level of contention. They emphasize the need for algorithms that automatically select protocols according to the level of contention. We would like to design reactive algorithms with performance that follows the ideal curves. As we will see in Section 3.5, our reactive algorithms perform very closely to this ideal.

3.2 The Design of Protocol Selection Algorithms

This section develops a framework for designing and reasoning about correct and efficient protocol selection algorithms. Recall that the two main issues in designing protocol selection algorithms are i) providing efficient *methods* for selecting and changing protocols, and ii) designing intelligent *policies* for deciding when to change protocols. This section addresses the first of these issues.

The difficulty of designing a protocol selection algorithm lies in the need to coordinate concurrent protocol executions with protocol changes. For performance reasons, the algorithm should allow protocol executions and changes to run concurrently. However, there needs to be some form of concurrency control to maintain correctness. Conceptually, the algorithm has to ensure that all synchronizing processes agree on which protocol to use in the face of dynamic protocol changes.

To ease the design effort, we would like to modify the existing protocols as little as possible. We present a set of conditions that a protocol selection algorithm should satisfy in order to preserve correctness, while allowing concurrent protocol executions and changes.

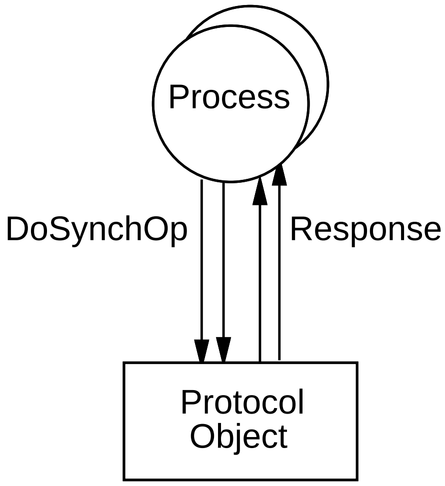


Figure 3.3: *A concurrent system model of a passive synchronization algorithm.*

We then introduce the notion of *consensus objects*[1] that help satisfy these conditions efficiently, with minor modifications to the original protocols.

3.2.1 Concurrent System Model

When designing a protocol selection algorithm, we are given a set of protocols that implement a synchronization operation. We assume that these protocols have been designed to handle concurrent synchronization requests correctly, according to the specification of the synchronization operation. Thus, we model a passive synchronization algorithm as a concurrent object that supports concurrent requests to perform synchronization operations, as illustrated in Figure 3.3. The concurrent object encapsulates the state of the synchronization protocol, and a process synchronizes by issuing a synchronization request (`DoSynchOp`) to the concurrent object.

In contrast to a passive algorithm that uses a single, fixed protocol, a reactive algorithm uses a protocol selection algorithm to select among multiple protocols. The protocol selection algorithm implements a concurrent object that supports operations to perform the synchronization operation and to change protocols. We model a protocol selection algorithm as a *protocol manager* and a set of concurrent *protocol objects*, as illustrated in Figure 3.4. Each protocol object represents a synchronization protocol and supports a set of operations that allows it to be selected by a protocol manager. We define these operations further below.

[1]These consensus objects are unrelated to Herlihy's wait-free consensus objects in [42].

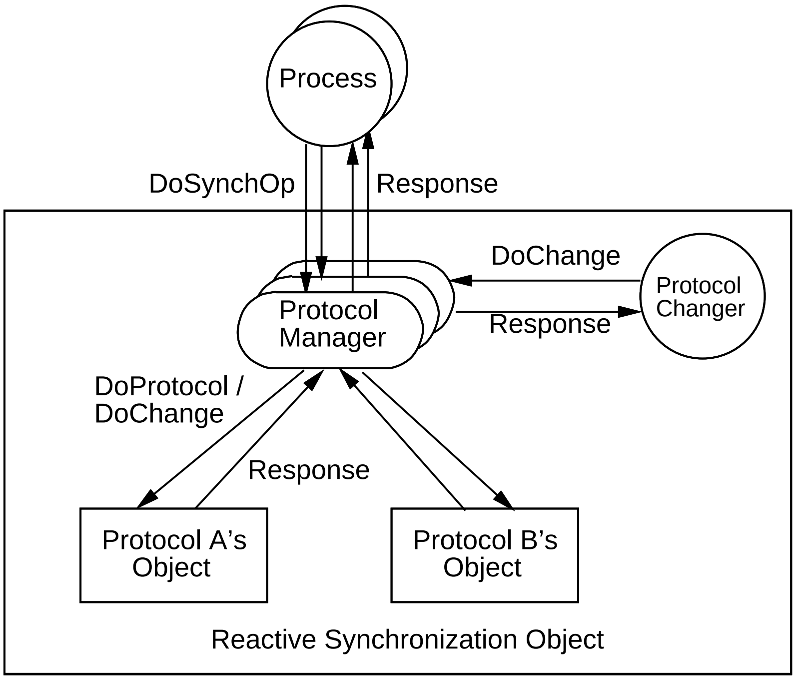


Figure 3.4: *A concurrent system model of a protocol selection algorithm.*

The protocol manager mediates concurrent access to the protocol objects and presents a conventional interface to the synchronizing processes. There are multiple instances of the protocol manager, one for each process. Informally, one can view the protocol manager as a procedure that is called by a synchronizing process. A synchronizing process issues a synchronization request (`DoSynchOp`) to the protocol manager. The protocol manager services the request by interacting with the protocol objects and returning the response from one of the protocol objects. A process may also request a protocol change by issuing a change request (`DoChange`) to the protocol manager. We model protocol changes as being generated by an internal process, although in principle any process can issue a change request. Again, the manager services the change request by interacting with the protocol objects. It is important to note that all communication occurs via the protocol objects: protocol managers do not communicate with each other. Thus, we can restrict the task of concurrency control to the protocol objects.

Execution Histories

We use the following notation for describing concurrent executions at a protocol object. A protocol manager executing on behalf of process P issues a *request* $(P, \texttt{op}, x)$ to object x to perform the operation named `op`. A protocol manager receives a *response* $(P, \texttt{res}, x)$

from object x, where `res` is the result value. We assume that process and object names are unique so that we can match requests and responses: a request matches a response if their process and object names agree.

A concurrent execution consists of an interleaving of requests and responses from multiple processes at each object. However, only a subset of these possible interleavings represent correct executions. To aid in describing correct executions, we follow the example of Herlihy and Wing in [23], and model a concurrent execution by a *history*.

A history is a finite sequence of request and response *events*. A *process history*, $H|P$, of a history H is the subsequence of events in H whose process names are P. Similarly, an *object history*, $H|x$, is the subsequence of events in H whose object names are x.

A history H represents an execution if each of its process histories $H|P$ follows the real-time order of requests and responses seen by each process P, and each of its object histories $H|x$ follows the real-time order of requests and responses seen by each object x. Two histories H and H' are *equivalent* if for every process P, $H|P = H'|P$.

A history is *sequential* if its first event is a request, and it alternates matching requests and responses. A history H is *well-formed* if each process history $H|P$ is sequential. In this model, we assume well-formed histories to capture the notion that a process represents a sequential thread of control.

A history H induces a partial order $\prec_H$ on its operations: $op_1 \prec_H op_2$ if $response(op_1)$ precedes $request(op_2)$ in the history. Two operations are *concurrent* if they are unordered by $\prec_H$.

In the following sections, we first provide a specification of a protocol object, and an implementation of a protocol manager that relies on that specification. We then address the problem of implementing a protocol object so as to satisfy its specification in the face of concurrency. We use pseudo-code to provide readable descriptions of the specifications and implementations. However, since pseudo-code can be somewhat imprecise, we also provide more precise specifications and implementations of protocol objects and managers using the Spec language [36] in Appendix B.

3.2.2 Protocol Object Specification

To allow a protocol manager to select and change protocols, we require that each protocol object supports operations to "validate" and "invalidate" itself, in addition to an operation to execute the synchronization protocol. The high-level idea is to allow the protocol manager to designate a single protocol as the protocol to be used for synchronization by validating

```
type prot_obj = record
    valid : boolean              // is protocol valid?
    RunProtocol : procedure      // runs the original protocol
    UpdateProtocol : procedure   // resets protocol to a consistent state
    state : prot_state           // state of the protocol

procedure DoProtocol(p : prot_obj) returns (V, invalid)
    if p.valid = true
        return p.RunProtocol(p)
    else
        return invalid

procedure Invalidate(p : prot_obj) returns boolean
    if p.valid = true
        p.valid := false
        return true
    else
        return false

procedure Validate(p : prot_obj)
    if p.valid = false
        p.UpdateProtocol(p)
        p.valid := true

procedure IsValid(p : prot_obj) returns boolean
    return p.valid
```

Figure 3.5: A specification of the operations of a protocol object.

one protocol and invalidating the others. To this end, we require that each protocol object supports the following operations:

`DoProtocol`	–	Performs the synchronization operation with the protocol associated with the protocol object.
`Invalidate`	–	Invalidates the protocol object.
`Validate`	–	Updates the protocol object to a consistent state and validates it.
`IsValid`	–	Returns "true" if the protocol object is valid, "false" otherwise.

In the following discussion, we term the `DoProtocol` operation as a *protocol execution*, and the `Invalidate` and `Validate` operations as *protocol change* operations.

In order to present a general framework that is independent of the specification of the synchronization operation to be implemented, we specify protocol objects operationally in terms of the original protocols. Figure 3.5 presents a specification of a protocol object in terms of the original protocol (via `RunProtocol`). We require that an implementation of a protocol object exhibits object histories that are equivalent to some *sequential* execution of the specification. The specification constrains the allowable histories of requests and responses at a protocol object, and any implementation of a protocol object is required to exhibit only a subset of these allowable histories.

3.2.3 A Concurrent Protocol Manager

A protocol manager provides two interface procedures, `DoSynchOp` and `DoChange`, for concurrent processes to execute the synchronization operation and to change the protocol in use, respectively. The responsibility of the protocol manager is to manipulate the protocol objects in order to synthesize the synchronization operation, while allowing concurrent protocol change requests. With the above specification of protocol objects, it is straightforward to implement a protocol manager: for correct execution, the protocol manager simply needs to satisfy the following two conditions:

1. It should return results only from valid protocol executions.
2. It should maintain the invariant that there exists at most one valid protocol object.

Figure 3.6 presents a protocol manager that relies on the specification of protocol objects presented above to satisfy the two requirements. The protocol manager selects between two protocols, `P1` and `P2`, although it should be straightforward to extend it to more than two protocols. The protocol manager uses `DoProtocol` to execute the protocol associated with the object, and `Validate` and `Invalidate` to change protocols.

To perform a synchronization operation, the protocol manager checks which protocol object is valid and executes the protocol associated with it. Since the check and the protocol execution are not an atomic unit, the protocol manager can never be sure that the protocol will remain valid throughout the execution. The protocol manager relies on the flag returned by `DoProtocol` to indicate whether the protocol execution was performed on a valid object. It loops until `DoProtocol` indicates a valid execution. In this way, the protocol manager returns results only from valid protocol executions.

To perform a protocol change, the protocol manager first attempts to invalidate a protocol. If it succeeds in invalidating a valid protocol, signified by `Invalidate` returning `true`, it validates the other protocol. This preserves the invariant that there exists at most one valid protocol, assuming that the system initially has only one valid protocol.

```
type prot_objs = record
    p1 : prot_obj   // protocol object for Protocol 1
    p2 : prot_obj   // protocol object for Protocol 2

procedure DoSynchOp(ps : prot_objs) returns V
    v : (V, invalid) := invalid
    repeat while v = invalid
        if IsValid(ps.p1)
            v := DoProtocol(ps.p1)
        else if IsValid(ps.p2)
            v := DoProtocol(ps.p2)
    return v

procedure DoChange(ps : prot_objs)
    if Invalidate(ps.p1)
        Validate(ps.p2)
    else if Invalidate(ps.p2)
        Validate(ps.p1)
```

Figure 3.6: A protocol manager.

3.2.4 Protocol Object Implementations

Now that we have an implementation of a protocol manager, we explore possible implementations of protocol objects. We begin with a straightforward approach that relies on locks to ensure that the history of concurrent operations at each protocol object is equivalent to some *sequential* execution of the specification. Unfortunately, the implementation suffers from poor performance and is unsuitable for implementing reactive locks. To improve performance, we exploit properties of the original protocols that allow concurrent operations. Specifically, we describe the notion of consensus objects that allow a protocol object to ensure that protocol changes are serializable.

A Naive Implementation

A simple approach to implementing a protocol object is to use locks to ensure that the protocol object operations execute atomically. This would immediately satisfy the sequential specification of a protocol object by serializing all operations. One can think of the locks as placing atomicity brackets around the body of each procedure in the specification.

```
procedure DoProtocol(p : prot_obj) returns (V, invalid)
    v : (V, invalid) := invalid
    Lock()
    if p.valid = true
        v := p.RunProtocol(p)   // run the original protocol
    Unlock()
    return v

procedure Invalidate(p : prot_obj) returns boolean
    b : boolean := false
    Lock()
    if p.valid = true
        p.valid := false; b := true
    Unlock()
    return b

procedure Validate(p : prot_obj)
    Lock()
    if p.valid = false
        p.UpdateProtocol(p)     // reset protocol to a consistent state
        p.valid := true
    Unlock()

procedure IsValid(p : prot_obj) returns boolean
    return p.valid
```

Figure 3.7: *A naive implementation of a protocol object based on locks.*

Figure 3.7 presents such an implementation. Unfortunately, there are several problems with this implementation that preclude it from being a practical implementation.

1. The use of locks around calls to `RunProtocol` have the undesired effect of serializing protocol executions, hurting the performance of protocols like software combining trees that rely on concurrent execution for performance.

2. Each synchronization operation now involves acquiring and releasing a lock in addition to executing a protocol. This may add a significant cost to the execution of a synchronization operation, especially if the protocol costs of the original protocols are comparable to the cost of acquiring and releasing a lock.

3. When designing a reactive algorithm for mutual-exclusion locks, the use of locks in

the protocol manager itself leads to the recursive problem of what protocols to use for these locks.

We could solve the first problem of unnecessarily serializing protocol executions by using reader-writer locks [14] instead of mutual-exclusion locks. `DoProtocol` would acquire a read lock, while `Invalidate` and `Validate` would acquire write locks. However, the other two problems persist.

3.2.5 Serializing Protocol Changes

The need to allow concurrent protocol executions at a protocol object brings up an interesting observation: an implementation of a protocol object needs to serialize only protocol change operations (`Validate` and `Invalidate`) with respect to other operations. That is, at each object, we only need to ensure that a protocol change operation never runs concurrently with any other operation. We can (and should) allow protocol executions to execute concurrently because the original protocols were designed to handle concurrent executions correctly. We term a concurrent execution that satisfies this restriction a $\mathcal{C}$*-serial* execution. ($\mathcal{C}$ for change.) $\mathcal{C}$-serial executions are modeled by $\mathcal{C}$-serial histories:

Definition 1 *A history H is $\mathcal{C}$-serial iff for each protocol object x, for all pairs of operations, op_1, op_2 in the object histories $H|x$,*

$$opname(op_1) = \texttt{Invalidate} \vee \texttt{Validate} \quad \Rightarrow \quad op_1 \prec_H op_2 \ \vee \ op_2 \prec_H op_1.$$

As an illustration, Figure 3.8 presents some example histories. History $H1$ is $\mathcal{C}$-serial since protocol changes at an object are serialized. History $H2$ is not $\mathcal{C}$-serial because $(Q, \texttt{Invalidate}, x)$ overlaps $(R, \texttt{DoSynchOp}, x)$. Finally, history $H3$ is $\mathcal{C}$-serial although $(Q, \texttt{Invalidate}, x)$ overlaps $(R, \texttt{DoSynchOp}, y)$ because they represent operations at different objects.

In a $\mathcal{C}$-serial execution, protocol changes partition the operations at each protocol object into phases of concurrent protocol executions. The motivation for restricting ourselves to $\mathcal{C}$-serial executions is that it allows the protocol selection algorithm to rely on the correctness of the original protocols for the correctness of concurrent protocol executions in between protocol changes.

Figure 3.9 illustrates an example history with the protocol manager of Figure 3.6 and multiple protocol objects that observe $\mathcal{C}$-serial histories. The protocol manager and the

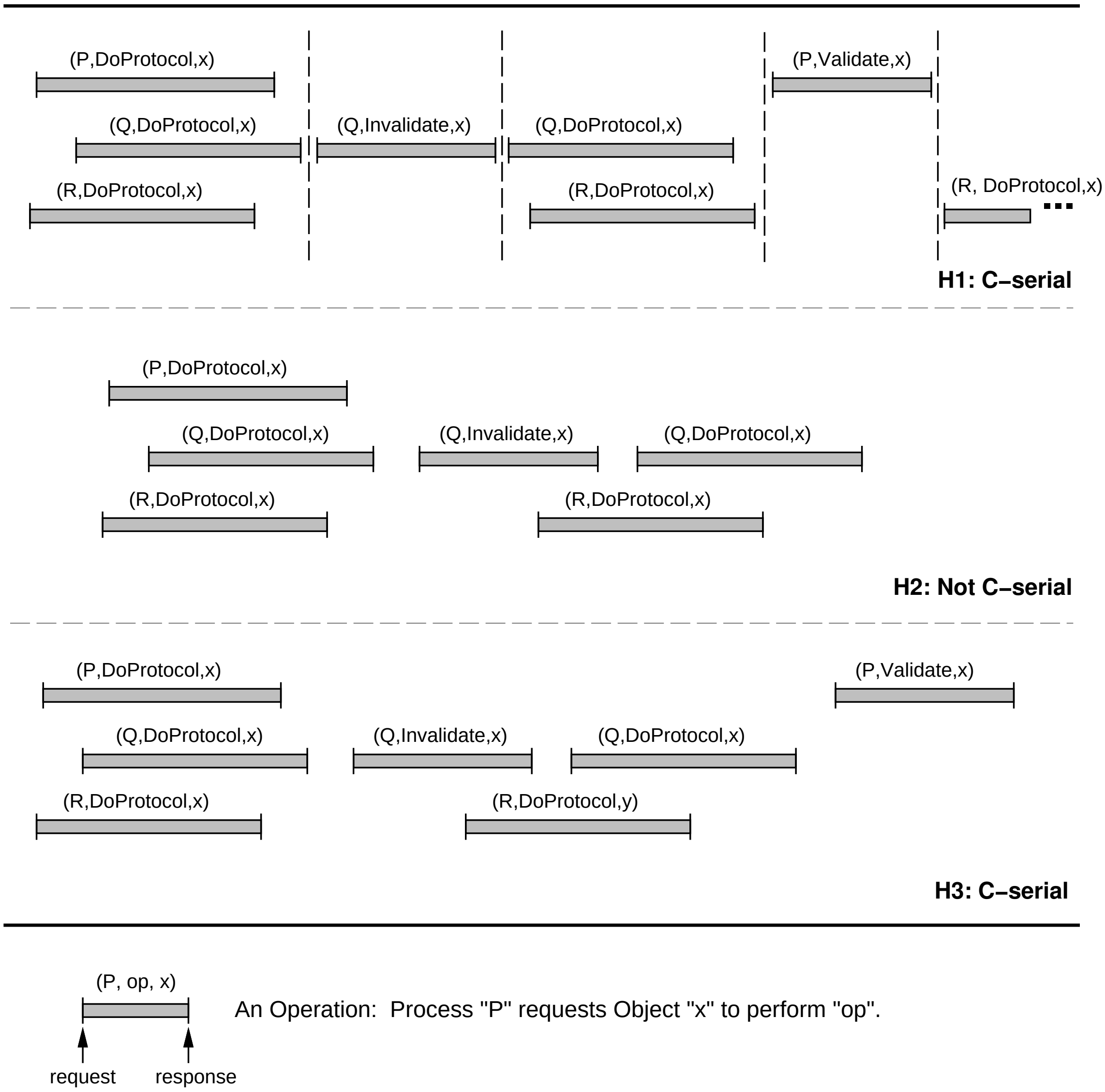


Figure 3.8: *Example $\mathcal{C}$-serial and non-$\mathcal{C}$-serial histories.*

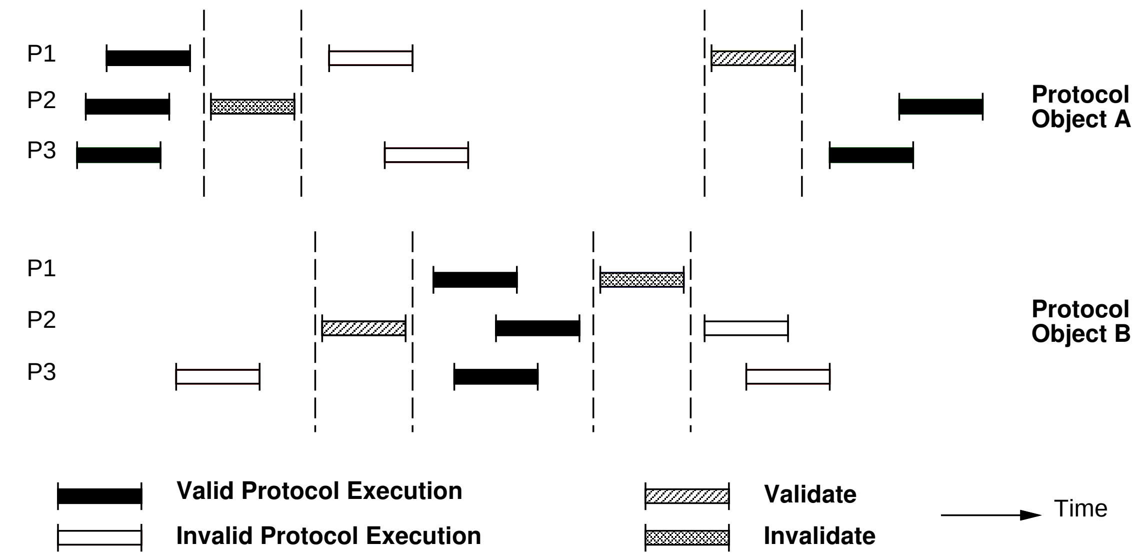


Figure 3.9: *Example $\mathcal{C}$-serial history with multiple protocol objects under the protocol manager. Initially, protocol A is valid. Process 2 performs a protocol change to protocol B. Process 1 then performs a protocol change back to protocol A.*

protocol objects cooperate to ensure that at most one protocol is valid at any time and that results are returned only from executions of valid protocols.

The concept of $\mathcal{C}$-serial executions allows us to relax our specification of protocol objects. Recall that in Section 3.2.2, we require an implementation to exhibit histories that are equivalent to some *sequential* execution of the protocol object specification in Figure 3.5. In the relaxed specification, we require an implementation to exhibit histories that are equivalent to some $\mathcal{C}$-serial execution of the specification. We term a history derived from a $\mathcal{C}$-serial execution of the specification a *legal* $\mathcal{C}$-serial history.

An implementation may allow concurrent protocol changes, as long as the protocol changes are *serializable* with all other operations, and protocol executions use the original protocol. Two concurrent operations are serializable if the behavior of the operations is "equivalent" to the behavior of executing the operations sequentially, one after the other. An implementation is $\mathcal{C}$-serializable if it satisfies these conditions. More formally,

Definition 2 *A protocol selection algorithm is $\mathcal{C}$-serializable if for each history H, there exists a legal $\mathcal{C}$-serial history H' such that*

(1) for all P, $H|P = H'|P$.

(2) $\prec_H \subseteq \prec_{H'}$.

Condition (1) ensures that H and H' are equivalent histories, and Condition (2) ensures that the equivalent legal $\mathcal{C}$-serial history, H', respects the real-time precedence ordering of the operations in H.

With a $\mathcal{C}$-serializable execution, a protocol execution that runs concurrently with a protocol change is well-defined: it is ordered either before or after a protocol change, and is either a valid or invalid execution.

A $\mathcal{C}$-serializable Implementation with Consensus Objects

We have defined $\mathcal{C}$-serializability as a correctness condition for implementing protocol selection algorithms. Now we need to address the question of how an implementation can enforce $\mathcal{C}$-serializable executions. As we observed, the explicit use of locks is unacceptable due to performance reasons. Fortunately, as we will see below, some protocols allow us to serialize protocol executions and changes quite naturally.

What properties of a protocol can we exploit for efficiently ensuring $\mathcal{C}$-serializable executions? It turns out that there is a property present in any protocol for locks, and also in combining tree protocols, that we can use to serialize protocol changes quite naturally.

Each of these protocols has the property that there is a unique object that some synchronizing process must access atomically, exactly once, in order to complete the protocol. We term such an object a *consensus object*. A process executing a protocol has to either access that protocol's consensus object, or communicate with some other process that accesses the consensus object.

For descriptive reasons, we subdivide the execution of such protocols into four phases: a *pre-consensus* phase, an *in-consensus* phase, a *wait-consensus* phase, and a *post-consensus* phase. Thus we can canonically describe the execution of a protocol with consensus objects as such:

```
procedure RunProtocol(p)
    if PreConsensus(p)              % pre-consensus phase
        AcquireConsensus(p);
        InConsensus(p);             % in-consensus phase
        ReleaseConsensus(p);
    else
        WaitConsensus(p);           % wait-consensus phase
    PostConsensus(p)                % post-consensus phase
```

In other words, each protocol execution either goes through the sequence `PreConsensus; AcquireConsensus; InConsensus; ReleaseConsensus; PostConsensus`, or the sequence `PreConsensus; WaitConsensus; PostConsensus`. `PreConsensus` returns `true` if a process should go to the in-consensus phase, or `false` if the process should go to the wait-consensus phase.

For example, in a combining-tree protocol, ascending the tree represents the pre-consensus phase, and descending the tree represents the post-consensus phase. A process that acquires exclusive access to the root is in the in-consensus phase, while a process that waits at the intermediate nodes is in the wait-consensus phase.

We can consider `AcquireConsensus` and `ReleaseConsensus` as acquiring and releasing a lock. However, this is not necessarily the case. As we will see when we investigate protocol selection algorithms for message-passing protocols, a process reaches in-consensus when executing inside an atomic message handler, and requires no locking. We only require that there exist a critical section to which we can add some code.

Besides the existence of a consensus object, we also require the protocols to satisfy the following properties:

1. A process in wait-consensus must be waiting for a process that reaches the in-consensus phase, possibly through some dependency chain of waiting processes.

2. Once a process has reached the post-consensus phase, it must be able to complete execution of the protocol, and is unaffected by future modifications to the consensus object.

3. A protocol can be updated consistently by a process that has atomic access to that protocol's consensus object.

These properties allow a protocol selection algorithm to execute and change protocols concurrently while maintaining $\mathcal{C}$-serializability. We mark a consensus object as valid or invalid to indicate whether its protocol is valid or not. Changing from protocol A to protocol B involves atomically accessing Protocol A's valid consensus object and invalidating it, then atomically accessing Protocol B's consensus object, updating the protocol, and validating its consensus object.

Since a protocol change involves acquiring atomic access to the consensus object, concurrent protocol changes are automatically serialized. Furthermore, since protocol executions have to access the consensus object in order to complete, concurrent protocol changes and protocol executions are also serialized. To see how this serialization occurs, consider a pair of concurrent protocol executions and protocol changes. The protocol execution can overlap the protocol change in the following two ways:

1. The protocol change overlaps a protocol execution that is in its pre-consensus phase. If the protocol change is an invalidate operation, the protocol execution will encounter an invalid consensus object and retry. If the protocol change is a validate operation, the protocol execution will terminate normally. In either case, the protocol execution behaves as if it is serialized *after* the protocol change.

2. The protocol change overlaps a protocol execution that is in its post-consensus phase. In this case, the protocol execution has already accessed the consensus object, and is unaffected by the protocol change. The protocol execution behaves as if it is serialized *before* the protocol change.

Figure 3.10 illustrates these two cases and shows how they are serialized.

Figure 3.11 presents an implementation of a protocol object for protocols with consensus objects. This implementation interfaces with the protocol manager in Figure 3.6, thus completing the implementation of the protocol selection algorithm. The implementation makes some minor modifications to the original protocol. It modifies `PostConsensus` to signal waiting processes (in the wait-consensus phase) that the protocol is valid. There is a new procedure, `PostConsensusFail`, that is identical to `PostConsensus` except that it signals waiting processes that the protocol is invalid. `WaitConsensus` returns false if it receives an invalid signal.

With this protocol object implementation, the protocol manager maintains the invariant that at most one protocol is valid by allowing only processes that have successfully invalidated a valid consensus object to validate another protocol. A process that executes an invalid protocol will either access that protocol's consensus object and notice that the protocol is invalid, or be informed of the fact through some other process that does. Upon detecting an invalid protocol, a process goes into the post-consensus phase to complete execution of the protocol and returns `false` to the protocol manager. The protocol manager will then retry the synchronization operation.

As a concrete example, we can apply the above framework to implement generic protocol selection algorithms for mutual-exclusion locks (mutexes) and reader-writer locks. Figures B.5 and B.6 in Appendix B present these algorithms.

Summary

We have presented a framework for designing correct and efficient protocol selection algorithms for protocols that exhibit consensus objects. The framework exploits the property of consensus objects to allow a process to execute any of the available protocols without

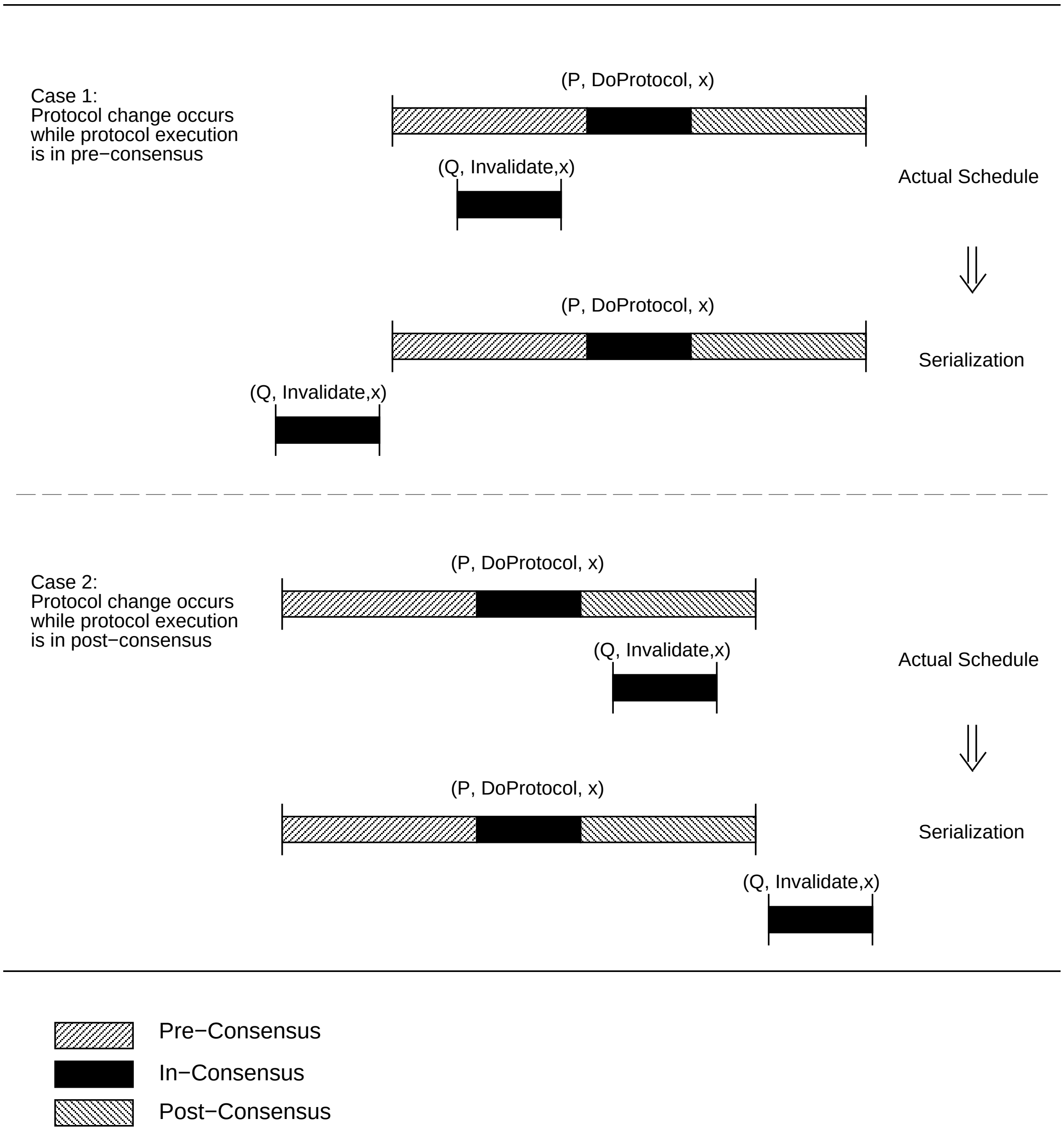


Figure 3.10: *Serializing protocol executions and changes with consensus objects.*

```
procedure DoProtocol(p : prot_obj) returns (V, invalid)
    if PreConsensus(p)
        AcquireConsensus(p)
        InConsensus(p)
        if p.valid
            ReleaseConsensus(p);
            return PostConsensus(p);
        else
            ReleaseConsensus(p);
            PostConsensusFail(p);
            return invalid
    else
        if WaitConsensus(p)
            return PostConsensus(p);
        else
            PostConsensusFail(p);
            return invalid

procedure Invalidate(p : prot_obj) returns boolean
    b : boolean := false
    AcquireConsensus(p)
    if p.valid
        p.valid := false; b := true
    ReleaseConsensus(p)
    return b

procedure Validate(p : prot_obj)
    AcquireConsensus(p)
    if p.valid = false
        p.UpdateProtocol(p)
        p.valid := true
    ReleaseConsensus(p)

procedure IsValid(p : prot_obj) returns boolean
    return p.valid
```

Figure 3.11: *An implementation of a protocol object for a protocol with consensus objects.*

prior coordination with other concurrent processes. The consensus objects ensure that concurrent protocol executions and protocol changes are serializable. Most importantly, the protocol selection algorithm adds very little overhead to the critical path of a protocol execution in the case where the protocol is valid. No extra synchronization is needed in this common case.

3.2.6 Performance Optimizations

While the above framework gives us a correct and efficient method for selecting protocols, there are a number of performance enhancing modifications that can be made. We did not include these enhancements in describing the framework because that would only obfuscate the discussion and detract the reader from the essential mechanisms that preserve correctness while allowing concurrent protocol changes. We discuss these performance enhancements here.

Inlining the protocol changer as part of protocol execution. In the system model, the protocol changer is modeled as a separate process that decides when to change protocols. Instead of relying on a separate process to initiate changes, we can make the protocol changer part of the protocol execution. Specifically, we can monitor run-time conditions as part of executing a protocol. This allows us to use otherwise idle spin-waiting cycles to perform useful work. For example, while spin-waiting for a test-and-set lock, a process can estimate contention levels by counting the number of failed attempts.

Combining the protocol manager with protocol executions. In the framework, the protocol manager and protocol executions were separate control structures. In particular, whenever an exceptional condition is detected, such as executing an invalid protocol, control has to return to the protocol manager. This interaction can be streamlined if we break the abstraction between the protocol manager and the protocol object and implement them in a single layer.

A mode variable for faster dispatching. The protocol manager checks the valid bit of each protocol to decide which protocol to execute. We can speed up this dispatch by using a mode variable to indicate which protocol is currently valid. The mode variable can be updated consistently during a protocol change. The use of a mode variable will allow faster dispatches, especially in the case where more than two protocols are being selected.

Moreover, on cache-coherent machines, the mode variable can be placed in a separate mostly-read cache line so that it can be read-cached by participating processors.

Protocol Specific Optimizations. The semantics of some protocols may allow us to further optimize the algorithms. One optimization that can be used when implementing reactive locks is to use the state of the lock to indicate whether the lock is valid or invalid. Instead of having a separate *valid* field, we can leave invalid locks in a locked state to indicate that they are invalid. The implementation of a reactive spin lock presented in the next section uses this trick to remove the need to check a *valid* field when using the test-and-test-and-set protocol.

3.3 Reactive Spin-Lock and Fetch-and-Op Algorithms

This section presents a high-level description of practical implementations of reactive spin locks and fetch-and-op that are largely based on the design framework outlined above. The algorithms have been further optimized with performance enhancements in order to minimize overheads in the steady-state case when protocol changes are not occurring. This section also describes how the reactive algorithms monitor run-time conditions to decide which protocol should be used. For further implementation details and an overview of the implementation process, refer to Section 3.7 and Appendix C.

3.3.1 The Reactive Spin-Lock Algorithm

The reactive spin-lock algorithm combines the low latency of a test-and-test-and-set lock with the scalability and fairness properties of the MCS queue lock by dynamically selecting between the test-and-test-and-set protocol and the MCS queue lock protocol. Figure 3.12 illustrates the components of the reactive lock. It is composed of two sub-locks (a test-and-test-and-set lock and an MCS queue lock), and a mode variable. Section 3.7 presents a pseudo-code listing of the reactive spin-lock algorithm. The algorithm uses the test-and-test-and-set spin lock itself as a consensus object, and the tail pointer of the MCS lock as a consensus object.

Intuitively, the reactive spin-lock algorithm works as follows. Initially, the test-and-test-and-set lock is free, the queue lock is busy, and the mode variable is set to `TTS`. A process acquires the reactive lock by acquiring either the test-and-test-and-set lock or the queue lock. The algorithm ensures that the test-and-test-and-set lock and queue lock are

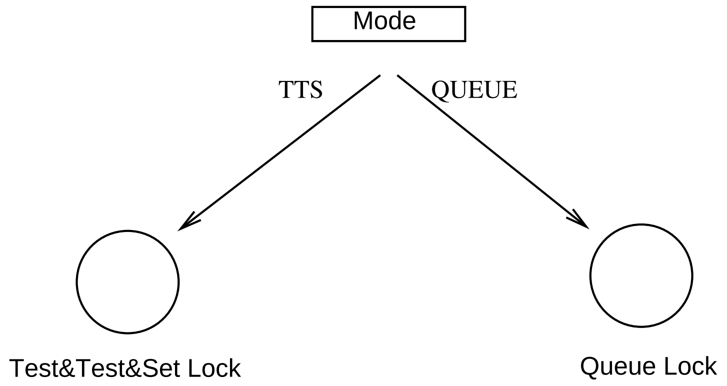


Figure 3.12: *Components of the reactive spin lock: a test-and-test-and-set lock, a queue lock, and a mode variable. The mode variable provides a hint to lock requesters on which sub-lock to use.*

never free at the same time, so that at most one process can successfully acquire a sub-lock. A process releasing the reactive lock can choose to free either the test-and-test-and-set lock or the queue lock, independent of which sub-lock it acquired.

The mode variable provides a hint on which sub-lock to acquire. On a cache-coherent multiprocessor, and assuming infrequent mode changes, the mode variable will usually be read-cached so that checking it incurs very little overhead. If the mode variable is `TTS`, a process attempts to acquire the test-and-test-and-set lock. Otherwise, if the mode variable is `QUEUE`, a process attempts to acquire the queue lock. To optimize for latency in the absence of contention, the reactive algorithm avoids checking the mode variable by optimistically attempting to acquire the test-and-test-and-set lock. The mode variable is checked only if the attempt fails.

The mode variable acts only a hint because there exists a race condition whereby the correct protocol to use can change in between when a process reads the mode variable and when that process executes the protocol indicated by the mode variable. However, since the reactive algorithm ensures that the test-and-test-and-set lock and queue lock are never free at the same time, processes that execute the wrong protocol will simply find the corresponding sub-lock to be busy. Such processes will either re-check the mode variable, or receive a retry signal while waiting, and retry the synchronization operation with a different protocol.

Changing protocols. We restrict protocol changes to a process that has successfully acquired a valid lock, thus ensuring atomicity while changing protocols. When changing from `TTS` mode to `QUEUE` mode, the reactive lock holder changes the value of the mode variable to `QUEUE`, then releases the queue lock, leaving the test-and-test-and-set lock in a busy state. When changing from QUEUE mode to TTS mode, the lock holder changes the value of the mode variable to TTS, signals any waiters on the queue to retry, then releases the test-and-test-and-set lock, leaving the queue lock in a busy state.

Detecting contention levels. The reactive algorithm estimates the level of lock contention to decide if the current lock protocol is unsuitable. In TTS mode, the reactive algorithm monitors the number of failed test-and-set attempts experienced by a process while acquiring the lock. This number indicates the number of times a process failed to acquire a lock after that lock was released, and is an indication of the level of lock contention. The reactive spin lock changes from the test-and-test-and-set protocol to the queuing protocol when the number of failed test-and-set attempts during a lock acquisition exceeds a threshold.

In QUEUE mode, the reactive algorithm monitors the number of times the MCS lock's queue is empty during a lock acquisition. Empty queues indicate low levels of lock contention. The reactive spin lock changes from the queuing protocol to the test-and-test-and-set protocol if a process finds the queue to be empty for some number of consecutive lock acquisitions.

3.3.2 The Reactive Fetch-and-Op Algorithm

The reactive algorithm for fetch-and-op chooses among the following three protocols:

1. A variable protected by a test-and-test-and-set lock.
2. A variable protected by a queue lock.
3. A software combining tree by Goodman *et al.* [15].

In the first two protocols, the consensus objects are the locks protecting the centralized variables. In the combining tree, the consensus object is the root of the combining tree. Appendix C presents a pseudo-code listing of the reactive fetch-and-op algorithm.

As in the reactive spin lock algorithm, a mode variable ushers processes to the fetch-and-op protocol currently in use. The reactive algorithm ensures that at most one of the fetch-and-op protocols' consensus objects is valid at any time. A process that accesses an invalid lock while executing one of the centralized protocols simply retries the fetch-and-op with another protocol. Unlike the reactive lock, we cannot optimistically try the test-and-test-and-set lock-based counter since this will have the effect of serializing accesses to the combining tree when contention is high, negating the potential benefits of parallelism in the combining tree.

For the combining tree, a process that accesses an invalid root has a set of processes it combined with that are waiting for a return value. These waiting processes are in the wait-consensus phase. Thus, the process that reaches the invalid root completes the combining tree protocol by descending the combining tree and notifying the processes that it combined with to retry the fetch-and-op operation. These waiting processes will in turn descend the tree and notify processes waiting farther down the tree to retry the operation.

Changing protocols. Only a process with exclusive access to the currently valid consensus object is allowed to change protocols. Recall that when changing to another protocol, the state of the target protocol needs to be updated to represent the current state of the synchronization operation. For fetch-and-op, the state of the synchronization operation is the current value of the fetch-and-op variable. In each of the three protocols, this state is represented by a variable that is modified only by processes with exclusive access to a protocol's consensus object. Thus the state of the target protocol can be easily updated by updating the value of this variable. As an optimization, the reactive algorithm keeps this variable in a common location so that updates are not necessary.

Detecting contention levels. As in the reactive spin lock, the reactive fetch-and-op algorithm changes from the test-and-test-and-set lock protocol to the queue lock protocol after some number of failed test-and-set attempts, and changes from the queue lock protocol to the test-and-test-and-set protocol if the queue lock's queue is empty for a number of successive fetch-and-op requests.

To decide whether to use the combining tree protocol, the reactive algorithm monitors the waiting time on the queue. Since the queue is FIFO, the waiting time on the queue provides a good estimate of the level of contention. If the waiting time exceeds a time limit, the algorithm changes from the queue lock protocol to the combining tree protocol.

To decide whether to return to the queue lock protocol from the combining tree protocol, the reactive algorithm monitors the number of combined requests in a process that reaches the root. This amounts to computing a fetch-and-increment along with the fetch-and-op, and seeing how large of an increment reaches the root. If the combining rate falls below a threshold, the reactive algorithm decides to switch back to the queue lock protocol.

3.4 Policies for Switching Protocols

In this section, we address the issue of designing intelligent policies for deciding when to switch protocols. First of all, a protocol selection algorithm has to determine if the protocol in use is not optimal by monitoring run-time conditions while executing a protocol. The available mechanisms for monitoring run-time conditions are protocol-specific, and were discussed in the preceding description of the reactive spin lock and fetch-and-op algorithms.

With an estimate of the run-time condition, a reactive algorithm must determine which protocol is optimal for that run-time condition. However, since the tradeoff between different protocols is architecture-dependent, the designer of a protocol selection algorithm needs to profile the execution of each protocol under different run-time conditions, as was done when compiling the results of Figure 3.2. Fortunately, this performance tuning is application-independent, and only needs to be done once per machine architecture.

Once a protocol selection algorithm has determined that another protocol is optimal, it needs to decide whether to switch to that protocol. Since switching protocols incurs a significant cost, this decision depends on the future behavior of run-time conditions, such as contention levels.

The default policy in our reactive algorithms is to switch protocols immediately upon detecting that the current protocol is not optimal. This has the advantage of tracking contention levels as closely as possible, but has the potential for pathologically bad performance. Specifically, contention levels can oscillate in such a way as to cause the policy to thrash and spend all its time switching protocols.

A more robust policy would avoid tracking temporary fluctuations in contention levels. A possible method is to compute a weighted average of the history of contention levels, and select the protocol that is optimal for the average contention level. Aging is a common operating system technique that can be used to compute the weighted average efficiently. Using a weighted average has the desired effect of ignoring temporary changes in contention levels.

Another possible approach is to use *hysteresis* to reduce the probability of thrashing among protocols. The idea here is to monitor the number of *consecutive* synchronization requests that have been serviced with a sub-optimal protocol. This constitutes a "bad streak", and the policy would be to switch protocols whenever the length of a bad streak exceeds a threshold. A larger threshold would result in more hysteresis.

The most promising approach we found is a policy that results in competitive performance. The policy is closely related to hysteresis. The idea is to maintain the *cumulative*

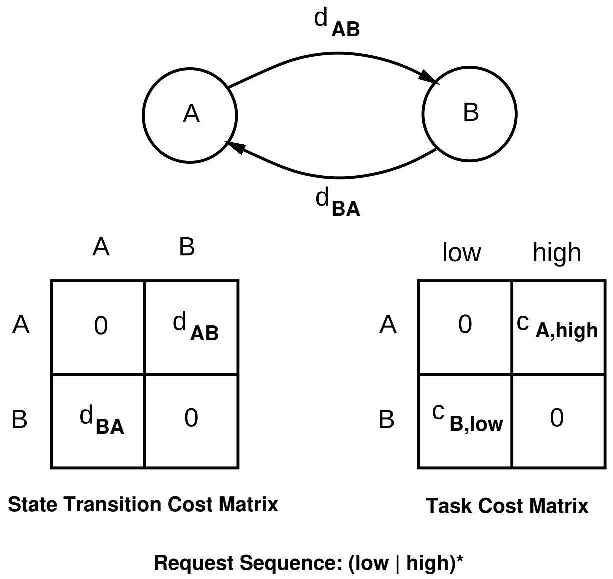


Figure 3.13: *A task system that offers a choice between two protocols, A and B.*

cost of using a sub-optimal protocol, and change protocols only when the cost exceeds a threshold. In contrast to hysteresis that requires an *unbroken* streak of sub-optimal protocol executions before switching protocols, this policy maintains the cumulative cost across breaks in streaks. We now describe this competitive policy in more detail.

3.4.1 A 3-Competitive Policy

We arrive at a 3-competitive policy for deciding when to switch between two protocols by mapping the problem onto a task system. (See Chapter 2 for a definition of task systems.) We model each protocol as a state in a two-state task system. The state transition cost matrix is composed of the costs of changing from one protocol to another. Since we are interested in the performance penalty of servicing a synchronization request with a sub-optimal protocol, the task cost matrix is composed of the *residual* costs of using a sub-optimal protocol over the optimal protocol. For example, if the optimal protocol costs 200 cycles to process a synchronization request while the sub-optimal protocol costs 500 cycles, then the residual cost of processing the request with the sub-optimal protocol is 300 cycles.

Figure 3.13 illustrates such a task system for two protocols. Protocol A is optimal under low contention, while Protocol B is optimal under high contention. d_{AB} is the cost of switching from protocol A to protocol B, while d_{BA} is the cost of switching from protocol B to protocol A. $C_{A,high}$ is the residual cost of servicing a high-contention synchronization

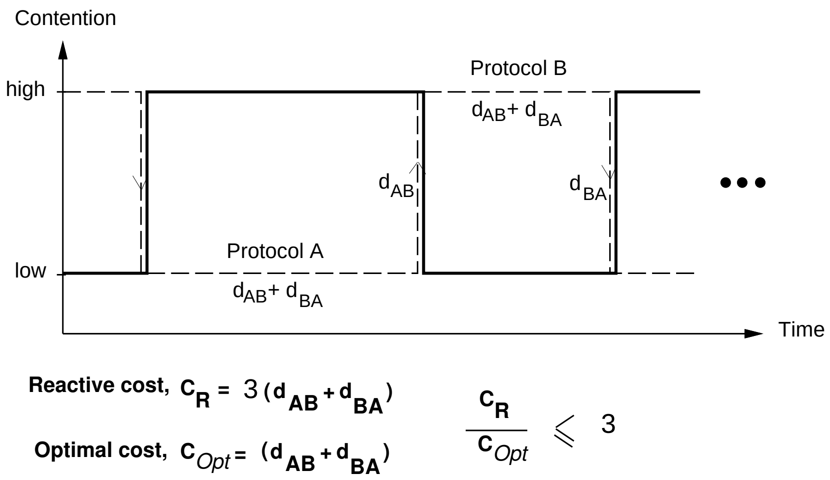


Figure 3.14: *Worst-case scenario for a 3-competitive protocol switching policy. The solid line plots the level of contention over time, and the dashed line indicates the protocol selected to satisfy the synchronization requests.*

request with Protocol A instead of Protocol B, while $C_{B,low}$ is the residual cost of servicing a low-contention request with Protocol B instead of Protocol A.

In [9], Borodin *et al.* present an algorithm for such a task system that has a competitive factor of $(2n-1)\psi(D)$. They term their algorithm a *nearly oblivious* algorithm. They also show that the competitive factor of $(2n-1)\psi(D)$ is a lower bound. $\psi(D)$ is the maximum ratio of the cost of traversing some cycle of a subset of task system states in one direction over the cost of traversing the same states in the other direction.

For a two-state system, $n = 2$ and $\psi(D) = 1$, and the nearly oblivious algorithm prescribes a state transition from state 1 to state 2 whenever the total task cost incurred since entering state 1 exceeds the total cost of changing to state 2 and back to state 1. This suggests that a protocol selection algorithm should switch from the current protocol to the other protocol whenever the cumulative residual cost of processing synchronization requests with the current protocol exceeds the cost of switching to the other protocol *and* back. Such a policy has a competitive factor of $2n - 1 = 3$.

To get an intuitive feel of why this policy results in 3-competitive performance, refer to Figure 3.14. The graph plots a worst-case scenario for a reactive algorithm that follows the switching policy described above. In this scenario, contention levels are chosen such that the reactive algorithm always uses the wrong protocol to service synchronization requests. As soon as the algorithm switches to a new protocol, the level of contention changes to favor the other protocol.

The residual cost incurred by the reactive algorithm over a period of two transitions is $3(d_{AB} + d_{BA})$. An optimal off-line algorithm would either choose to track the contention level and switch protocols twice, or choose not to switch protocols at all. In either case, it would incur a residual cost of $(d_{AB} + d_{BA})$. Thus the reactive algorithm incurs a residual cost that is at most three times that of an optimal off-line algorithm.

3.5 Experimental Results

To demonstrate the benefits of dynamic protocol selection, we measure the performance of the reactive spin lock and fetch-and-op algorithms that we designed in the previous section, and compare them with the best known passive algorithms. These measurements are mostly obtained from the Alewife simulator. We also present measurements from a 16-processor Alewife machine prototype to validate the simulation results.

There are two performance characteristics of reactive algorithms that we would like to investigate. First, for a given level of contention, we would like to see how close a reactive algorithm can approach the performance of the *best* static choice of protocols for that level of contention. This will indicate how small (or large) the run-time overhead of the reactive algorithm is, once it has settled down on the optimal protocol to use.

Second, when contention levels vary over time, a reactive algorithm may be able to *outperform* a passive algorithm. This will depend on the overhead of switching protocols, and on how frequently protocol changes are needed. If contention levels vary infrequently in comparison to the overhead of changing protocols, then a reactive algorithm will outperform a passive algorithm. However, time-varying contention levels are a double-edged sword. If contention levels vary too frequently, a reactive algorithm may thrash and perform badly.

To investigate the performance characteristics of reactive algorithms, we ran the following types of experiments.

Baseline Test This test precisely characterizes the cost of a synchronization algorithm for a given level of contention. It fixes the level of contention and measures the resulting cost of an algorithm in terms of processor cycles. This test exposes the overhead of detecting which protocol to use in a reactive algorithm. However, it does not expose the cost of changing protocols.

Multiple Object Test A typical parallel application has multiple synchronization objects, each with potentially different levels of contention. This test demonstrates the ability

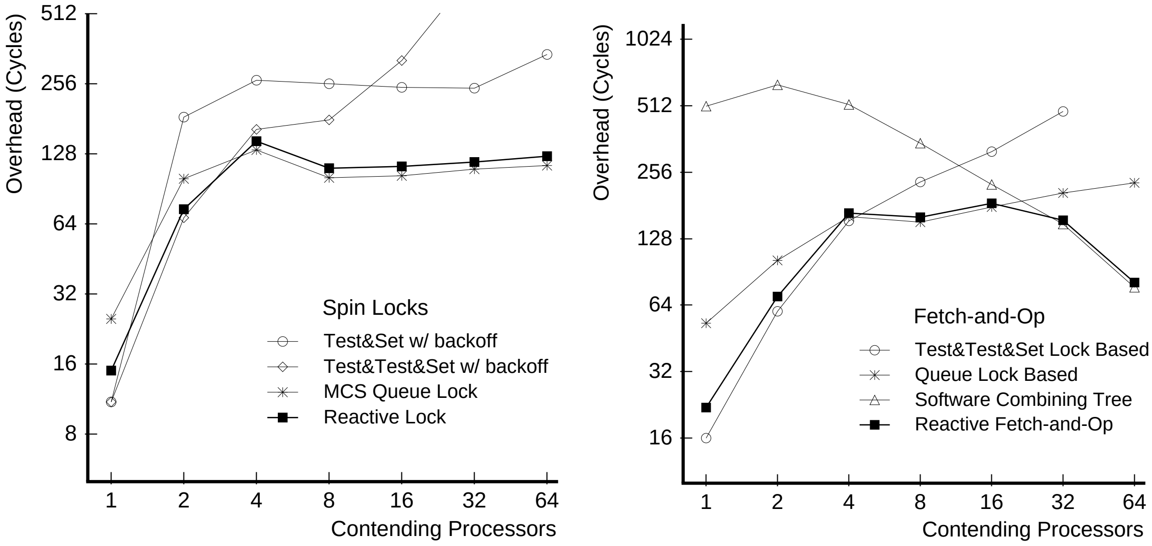


Figure 3.15: *Baseline performance of spin lock and fetch-and-op algorithms. "Overhead" represents the average number of cycles per completed synchronization operation that is due to the synchronization algorithm in use.*

of a reactive algorithm to select the best protocol for each synchronization object in a program.

Time-Varying Contention Test The level of contention at a synchronization object in a parallel application may vary over time. This test investigates the performance of reactive algorithms under time varying contention levels. It exposes the overhead of changing protocols and also the benefit of adapting to changing contention levels. Since this test requires only 16 processors, we ran it on the 16-processor Alewife prototype.

Application Measurements We implemented and ported several parallel applications that use spin locks and fetch-and-op in order to measure the effect of using different synchronization algorithms on the running time of applications.

3.5.1 Baseline Test

In this experiment, we compare the performance of the synchronization algorithms by measuring the *average overhead per synchronization operation* incurred by an algorithm at different levels of contention. This overhead represents the number of cycles in between successive synchronization operations at a single synchronization object that is due to the synchronization algorithm in use.

We derive the overhead by first measuring the average elapsed time in between successive synchronization operations. A fraction of this time is due to the latency introduced by the test loop itself. This test-loop latency can be computed from the parameters of the test loop, *e.g.*, the length of the critical section in the test loop. To arrive at the overhead due to the synchronization algorithm, we subtract the test-loop latency from the measured time.

The motivation for filtering out the test-loop latency from the average time in between successive synchronization operations is to make the results less dependent on the parameters of the test loop, and to focus on the overhead due to the synchronization protocol in use.

Figure 3.15 compares the baseline performance of the algorithms. The results show that the reactive algorithms succeed in selecting the best protocol to use, and are close to the performance of the best passive algorithms at all levels of contention.

Spin Locks Each processor executes a loop that acquires the lock, executes a 100-cycle critical section, releases the lock, and delays for a random period between 0 and 500 cycles.

```
procedure test_lock (l : ^lock)
    repeat while true
        lock(l)
        delay(100)               // critical section
        unlock(l)
        delay(random(0, 500))    // think time
```

The 100-cycle critical section models a reasonably small critical section when contention is involved: protected data has to migrate between caches, and it takes about 50 cycles to service a remote cache miss. The delay between lock acquisitions models some computation (think time) in between accesses to the lock, and it forces the lock to migrate between caches when there is contention. Otherwise, a single processor may hog the spin lock when using the test-and-set or test-and-test-and-set protocols, unfairly favoring their performance. This test program is similar to that used by Anderson [5].

Each data point represents the average lock overhead per critical section with P processors contending for the lock. To arrive at this measure, we first compute the average elapsed time per critical section by dividing the actual elapsed time by the number of critical sections. We then derive the test-loop latency per critical section from the length of the critical section (100 cycles) and the average think time (250 cycles). The test-loop latency should be 350 cycles at the one-processor data point, 175 cycles at the two-processor data point, and 100 cycles at the four-processor data point and beyond.

The average lock overhead per critical section is the difference between the average elapsed time and the test-loop latency per critical section. One can view the overhead as the number of cycles the spin lock algorithm adds to each critical section. Without contention, the average lock overhead represents the latency of an acquire-release pair. With contention, the average lock overhead represents the time to pass ownership of the lock from one process to another.

The results indicate that the reactive algorithm succeeds in selecting the test-and-test-and-set protocol at the one- and two-processor data points, and the queuing protocol at all other data points. They also show that the reactive algorithm adds very little overhead over statically selecting the best protocol.

Fetch-and-Op We use *fetch-and-increment* as a representative of a combinable fetch-and-op operation. Each processor executes a loop that executes a fetch-and-increment (a combinable instance of fetch-and-op), then delays for a random period between 0 and 500 cycles.

```
procedure test_fetch_and_incr (c : ^counter)
    repeat while true
        fetch_and_increment(c)
        delay(random(0, 500))    // think time
```

As in the baseline test for spin locks, the delay between increment requests forces the fetch-and-increment variable to migrate between caches when there is contention. We used a radix-2 combining tree with 64 leaves for the combining-tree protocol.

Each data point represents the average overhead per fetch-and-increment operation with P processors contending for the operation. To arrive at this measure, we first compute the average elapsed time per fetch-and-increment by dividing the actual elapsed time by the number of increments. We then compute the test-loop latency per increment, given zero-overhead fetch-and-increment operations. This latency is $250/P$ cycles, where 250 is the think time, and P is the number of contending processors. The average overhead per fetch-and-increment operation is the difference between the average elapsed time and the test-loop latency. One can view the overhead as the number of cycles the fetch-and-op algorithm adds to the generation of each increment.

The results indicate that the reactive algorithm succeeds in selecting the test-and-test-and-set lock-based protocol for the one- and two-processor data points, the queue-based protocol for the 4–16 processor data points, and the combining tree protocol for the 32- and

64-processor data points. The results also show that it is crucial to have a reactive algorithm for fetch-and-op. There exists a difference of several orders of magnitude in between the centralized and combining tree protocols. The reactive fetch-and-op algorithm combines the advantages of each of its component protocols: it has low latency when contention is low and high throughput when contention is high.

3.5.2 Spin Locks on Alewife Hardware

To verify the results from the simulator, we ran the baseline test for spin locks on the 16-processor Alewife prototype. The test is similar to the baseline test on the simulator. Each processor executes a loop that acquires a lock, executes a 100-cycle critical section, releases the lock, and delays for 250 cycles. Each processor loops 1,000,000/P times. Unlike the test on the simulator, we didn't use a random delay because computing `rand()` on Alewife consumes a few hundred cycles in itself. This is primarily due to the fact that Sparcle does not have hardware instructions for integer multiply and divide. On the simulator, we had the luxury of escaping to the simulator to compute the random delay.

We compute the average time per critical section by dividing the actual elapsed time by 1,000,000. We then subtract the ideal time per critical section (see description of baseline test for spin locks above) from the average time to arrive at the average lock overhead per critical section.

Figure 3.16 presents the results of this test. Again, we can see the contention-dependent tradeoff between the test-and-set lock and the MCS queue lock. We also see that the reactive lock manages to track the performance of the best protocol at different levels of contention. There are a couple of differences compared to the simulation results that we explain here.

The first difference is that the test-and-set lock performs better than predicted by the simulations with two processors contending for the lock. This difference is due to the unfairness property of the test-and-set lock. In the simulations, we observed that with two processors, the test-and-set lock is initially fair, and the lock bounces back and forth between the processors. However, eventually one of the processors gains control of the lock and the other processor simply backs off to the maximum delay. This has the effect of lowering the test-and-set lock overhead since the lock is no longer migrating between processors for each critical section. In the simulations, we measured lock overhead only during the initial period when the lock is fairly shared among the processors, while on the real hardware, we measured lock overhead for the entire duration of the test.

The second difference is that the lock overhead under high contention is smaller on the real machine than predicted by the simulation. This lock overhead represents the time it

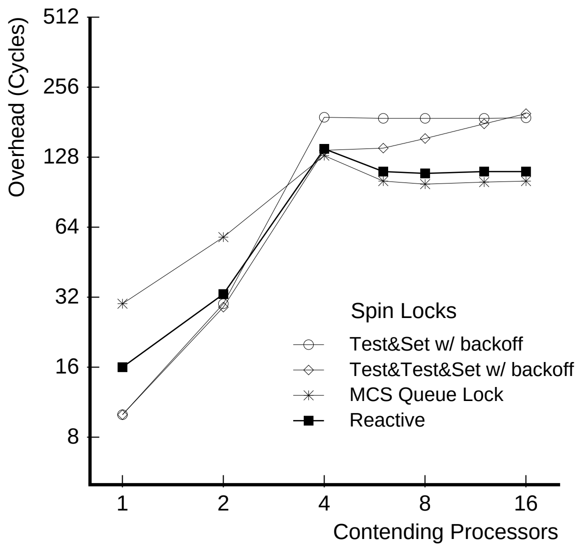


Figure 3.16: *Baseline performance of spin lock algorithms on the 16-processor Alewife prototype.*

takes to pass a lock from one processor to another. The reason for the lower overhead is that the Alewife prototype currently runs at 20MHz while the simulations assume a clock speed of 33MHz. Because Alewife uses an asynchronous network, and we measure overhead in terms of processor cycles, communication latencies appear shorter on the Alewife hardware.

Despite these minor differences, the results confirm that the reactive lock succeeds in choosing the right protocol at different levels of contention on the actual Alewife hardware.

3.5.3 Multiple Object Test

The baseline performance figures measure the performance of the reactive synchronization algorithms for a given level of contention at a single synchronization object. In practice, a parallel program may have multiple synchronization objects, each with different levels of contention. Since a reactive algorithm should select the best protocol to use at each of these objects, we can expect it to outperform a passive algorithm that uses the same protocol across all the objects.

To demonstrate this feature of reactive algorithms, we use a synthetic benchmark with 64 processors attempting to acquire and release a set of spin locks. We statically predetermine the contention level at each spin lock by assigning each of the 64 processors to one of the spin locks.

Each processor executes a loop that acquires a lock, increments a double precision floating point value associated with the lock, releases the lock, and delays for a random period between 0 and 500 cycles. Thus, the loop is identical to the one in the baseline test, except the critical section represents some actual computation instead of a fixed delay of 100 cycles. We measure the time for the processors to perform a total of 16,384 lock acquisitions and releases.

We compare four synchronization algorithms: i) a test-and-set spin lock algorithm, ii) an MCS queue lock algorithm, iii) our reactive spin lock algorithm, and iv) a simulated optimal algorithm. The simulated optimal algorithm queries the simulator for the best protocol to use at each lock. From the baseline results, we know that the test-and-set lock is optimal with less than four contending processors, while the MCS queue lock is optimal with four or more contending processors. The simulated optimal algorithm does not perform any run-time monitoring of contention levels nor does it attempt to perform any protocol changes. It provides a measure of how well an optimal static choice of protocols might perform, modulo the overhead of a conditional branch for querying the simulator at each synchronization operation.

Figures 3.17–3.19 present the results of running this test on a set of 12 different contention patterns. We illustrate each contention pattern as a histogram of lock contention. For example, Pattern 1 has one lock with 32 processors contending for it, and 32 locks with only one processor contending for each of them. Pattern 2 has two locks, each with 16 processors contending for it, and 32 locks with only one processor contending for each of them. The elapsed times are normalized to the simulated optimal algorithm.

The results show that when there is a mix of low and high contention locks, the reactive algorithm is able to outperform a passive algorithm that uses the same protocol for all of the locks. We can also see that it is difficult to predict from the mix of contention levels which passive algorithm to use. The reactive algorithm automatically selects the best protocol to use at each lock and performs within 8% of the simulated optimal algorithm.

It is interesting to consider the performance of the MCS lock under Patterns 5–8. Patterns 5–8 are similar to Patterns 1–4 except that the low-contention locks have two processors contending for them instead of one. Thus, we might expect the relative performance of the MCS lock to improve for Patterns 5–8 over Patterns 1–4. However, its performance actually degrades for Patterns 5–8. This is due to a race condition that inflates the cost of the MCS protocol under conditions of low, but non-zero contention.

This race condition occurs when a process releasing a lock sees that it has no successors and proceeds to empty the queue. However, before the queue is emptied, another process

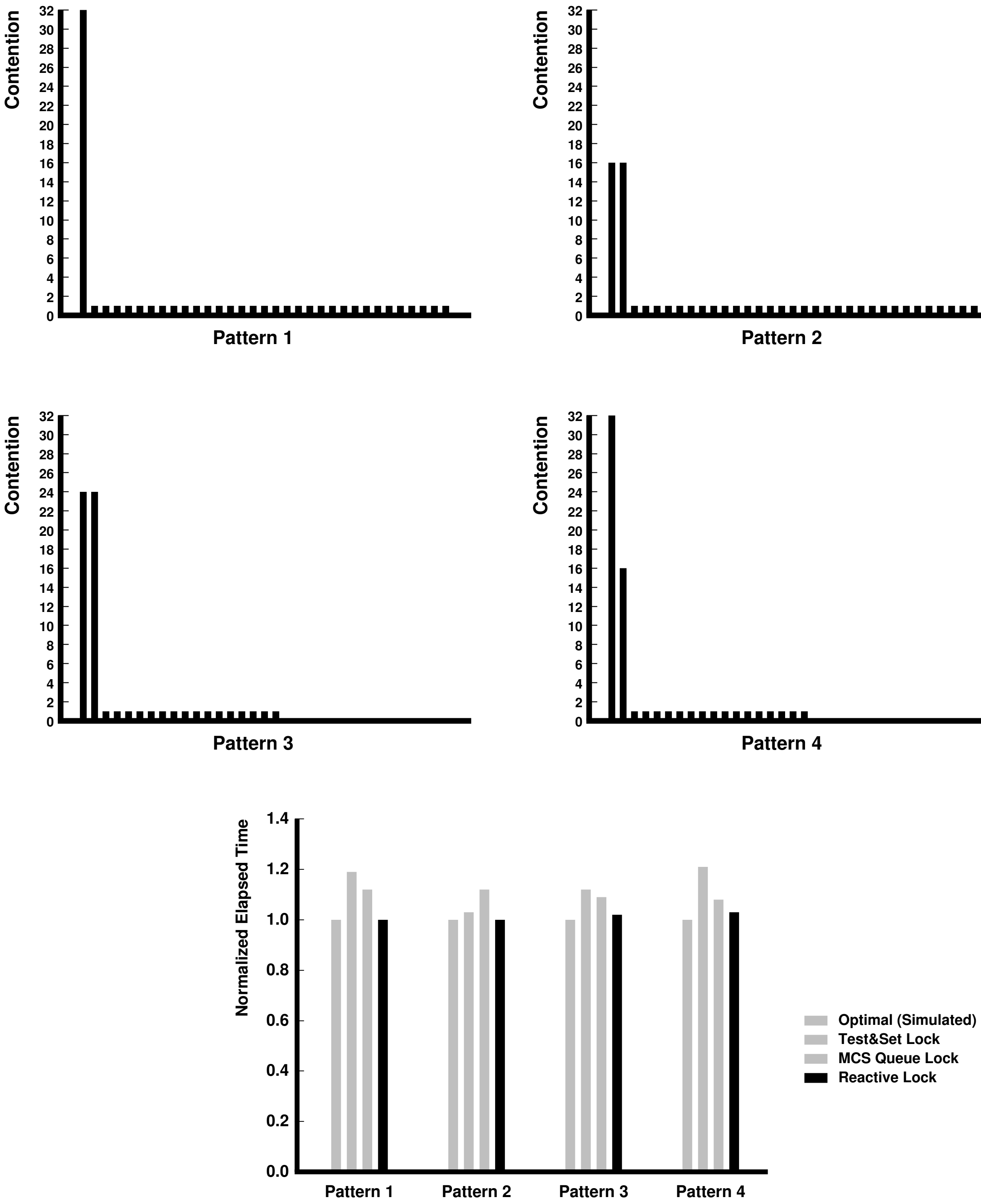


Figure 3.17: *Normalized elapsed times for the multiple lock test, contention patterns 1–4.*

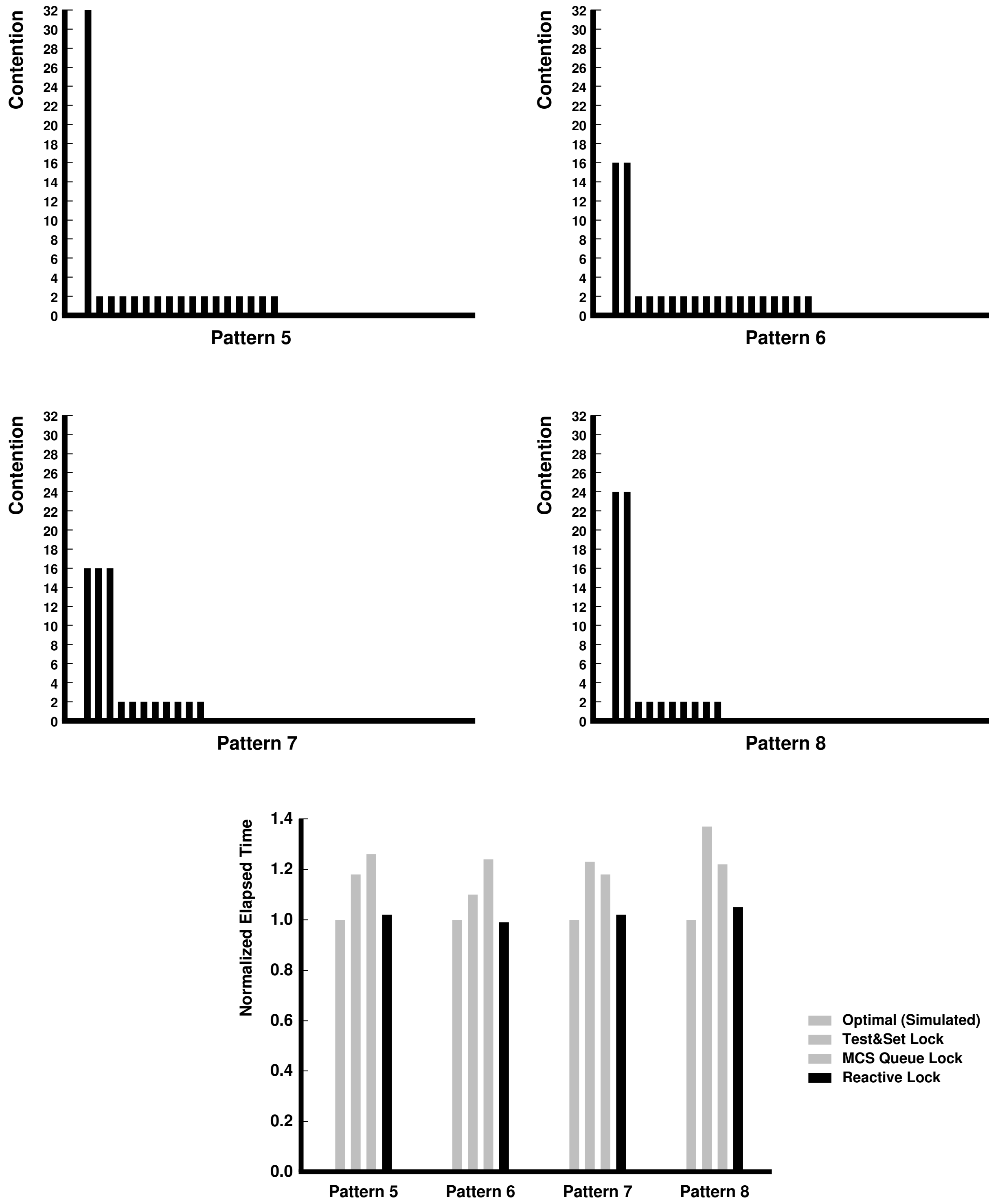


Figure 3.18: *Normalized elapsed times for the multiple lock test, contention patterns 5–8.*

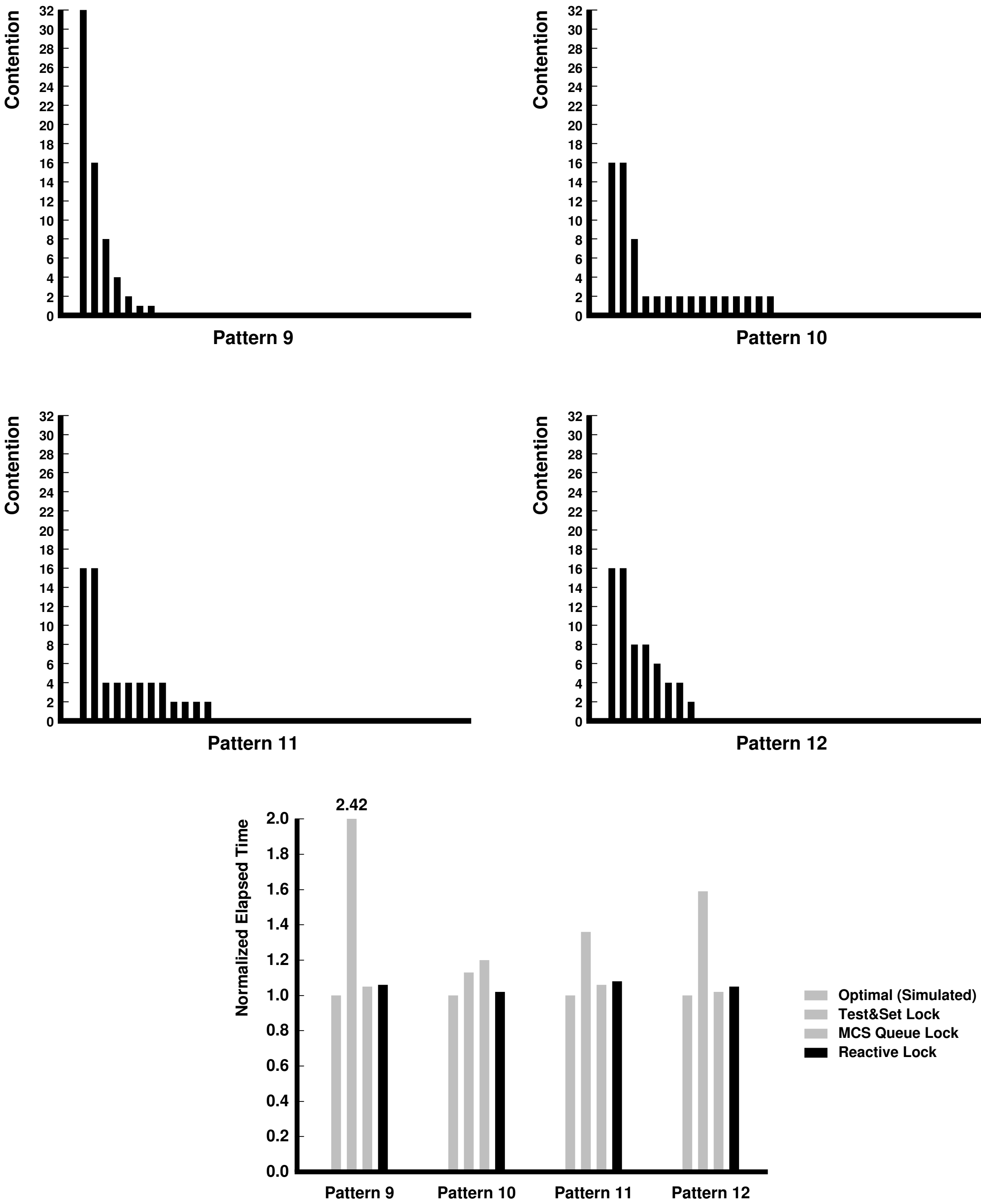


Figure 3.19: *Normalized elapsed times for the multiple lock test contention patterns 9–12.*

arrives and adds itself to the end of the queue, thereby violating the releasing process's assumption that no other processes are waiting for the lock. If this occurs, the MCS protocol executes some clean-up code that restores the queue.

In this test, the race condition occurs when there are two processors contending for a lock, and Patterns 5–8 have a large number of locks with two contending processors. This leads to the unexpectedly poor performance of the MCS queue lock. By selecting the test-and-set protocol when two processors are contending for the lock, the reactive algorithm avoids this pitfall.

3.5.4 Time-Varying Contention Test

While the preceding tests expose the run-time overhead of selecting protocols, they do not expose the overhead of changing protocols. To expose this overhead, we ran a test program that periodically switches between phases of no contention and high contention. Besides demonstrating the benefits of adapting to the level of contention, it also shows how badly a reactive algorithm might perform with frequently changing contention levels.

Figure 3.20 illustrates how the level of contention varies during the test. In the low-contention phase, a single processor executes a loop that acquires the lock, executes a 10-cycle critical section, releases the lock, and delays for 20 cycles. In the high contention phase, 16 processors concurrently execute a loop that acquires the lock, executes a 100-cycle critical section, releases the lock, and delays for 250 cycles. We measure the time for the test program to execute 10 periods. Since this test requires only 16 processors, we ran this test on the 16-processor Alewife prototype.

We compare the performance of our reactive lock with the test-and-set lock and the MCS lock. Figure 3.21 presents the results of this experiment, normalized to the execution time for the MCS lock. We vary the test program along two dimensions: i) the length of a period, measured as the number of locks per period, and ii) the percentage of locks that are acquired under high contention (% contention). The length of a period controls how frequently the reactive lock may have to switch protocols. Each period should cause the reactive lock to switch protocols twice.

First, consider the case when contention levels do not vary too frequently (towards the right end of each graph). As expected, the results show that the test-and-set lock outperforms the MCS queue lock when contention is rare (10% contention). When contention dominates (90% contention), the MCS queue lock outperforms the test-and-set lock. In both cases, the reactive lock approaches the performance of the better of the two passive algorithms. When there is some mix of low and high contention (30% contention), neither the test-and-set lock

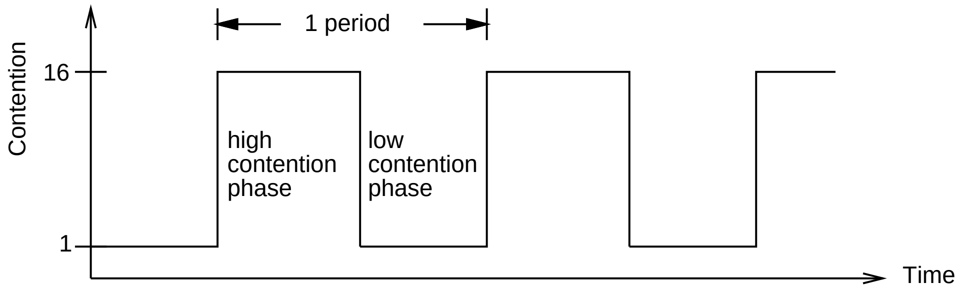


Figure 3.20: *The time-varying contention test periodically varies the level of contention to force the reactive lock to undergo protocol changes.*

nor the MCS queue lock has a clear advantage over the other. By continuously selecting the better protocol, the reactive lock outperforms both the test-and-set and MCS queue locks.

Next, consider the case when contention levels vary frequently (towards the left end of each graph). The results show that the overhead of switching protocols dominates and the performance of the reactive lock suffers. In the experiments, the performance of the reactive spin lock begins to deteriorate when forced to change protocols as frequently as every 1000 critical sections. However, the reactive lock is still always better than the worst static choice of protocols even under such extreme circumstances. It is interesting to note that the performance of the test-and-set protocol deteriorates when contention levels change frequently. This is because the test-and-set protocol does not handle bursty arrivals of lock requesters as well as the MCS queue lock protocol.

3.5.5 Alternative Switching Policies

The policy in the reactive spin-locks and fetch-and-op thus far has been to switch protocols immediately after detecting that it should be using another protocol. If contention levels vary across protocol breakeven points too frequently, this policy might cause a reactive algorithm to thrash and switch protocols needlessly. While we do not expect contention levels to vary so frequently in practice, we might want to use more intelligent policies for switching protocols to protect against such pathological behavior. We described several such policies in Section 3.2. Here, we use the time-varying contention test to measure the effect of using competitive techniques and hysteresis to decide when to switch protocols.

Recall that the competitive algorithm switches protocols after the cumulative cost of using the sub-optimal protocol exceeds the cost of switching to the other protocol and back. For the reactive spin lock, we assume a cost of 150 cycles for using the test-and-test-and-set protocol under high contention, and a cost of 15 cycles for using the MCS queue

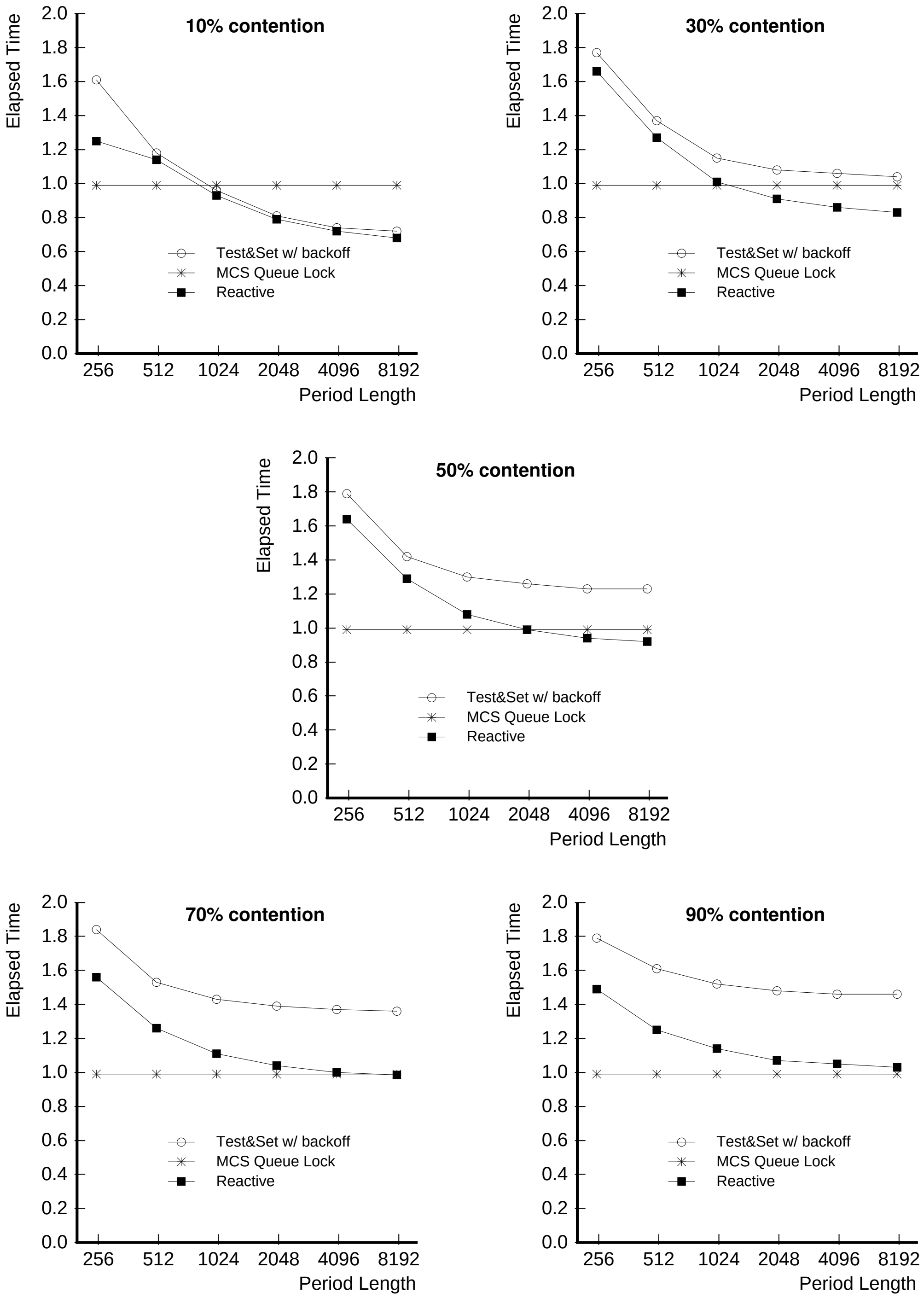


Figure 3.21: *Elapsed times for the time-varying contention test, normalized to the MCS Queue Lock.* Period Length *is measured as the number of locks acquired per period.*

lock protocol under low contention. The cost of changing from the test-and-test-and-set protocol to the MCS protocol is empirically observed to be about 8000 cycles, while the cost of changing from the MCS protocol to the test-and-test-and-set protocol is about 800 cycles. With these parameters, we implemented a 3-competitive policy for deciding when to switch protocols in the reactive spin-lock. The policy switches protocols whenever the cumulative cost of being in a sub-optimal protocol exceeds 8800 cycles.

Figure 3.22 presents the results of using a 3-competitive policy for switching spin-lock protocols in the time-varying contention test. The curve labeled Reactive, Always is the default policy of switching immediately upon detecting that it is using the sub-optimal protocol. The curve labeled Reactive, 3-competitive is the 3-competitive policy described above.

The results show that the competitive algorithm improves the performance of the reactive lock when switching frequencies are high, especially when contention is predominantly high (*cf.* curves for 70% and 90% contention with a period length of 256 locks per period). However, this comes at the price of lower performance at intermediate switching frequencies. As expected, at low switching frequencies (towards the right end of the graphs), the switching policies do not make much difference, except to add some constant overhead.

An alternative switching policy we explore is using hysteresis to reduce the probability of thrashing between protocols. Figure 3.23 presents the results of this experiment. We experimented with several settings of hysteresis levels. We use the following notation to describe the hysteresis levels: Hysteresis(x, y) means that the algorithm switches from the test-and-test-and-set protocol to the MCS protocol after x consecutive high-contention lock requests, and switches from the the MCS protocol to the test-and-test-and-set protocol after y consecutive low-contention lock-requests.

We measured the performance of the reactive spin lock algorithm under Hysteresis(20, 55), Hysteresis(500, 4), and Hysteresis(4, 500). Hysteresis(20, 55) matches the switching thresholds of the 3-competitive algorithm. Recall that the policy of hysteresis differs from the 3-competitive algorithm only in not maintaining the cumulative cost of servicing requests across breaks in bad streaks. Hysteresis(500, 4) and Hysteresis(4, 500) favor using the test-and-test-and-set protocol and MCS protocol, respectively.

The results were rather disappointing. Maintaining hysteresis adds a significant amount of run-time overhead to the reactive algorithm. The main problem is that the reactive algorithm always incurs an overhead to maintain statistics for hysteresis, even when the currently selected protocol is the optimal protocol. Contrast this with the 3-competitive algorithm that incurs an overhead to maintain statistics only when using a sub-optimal

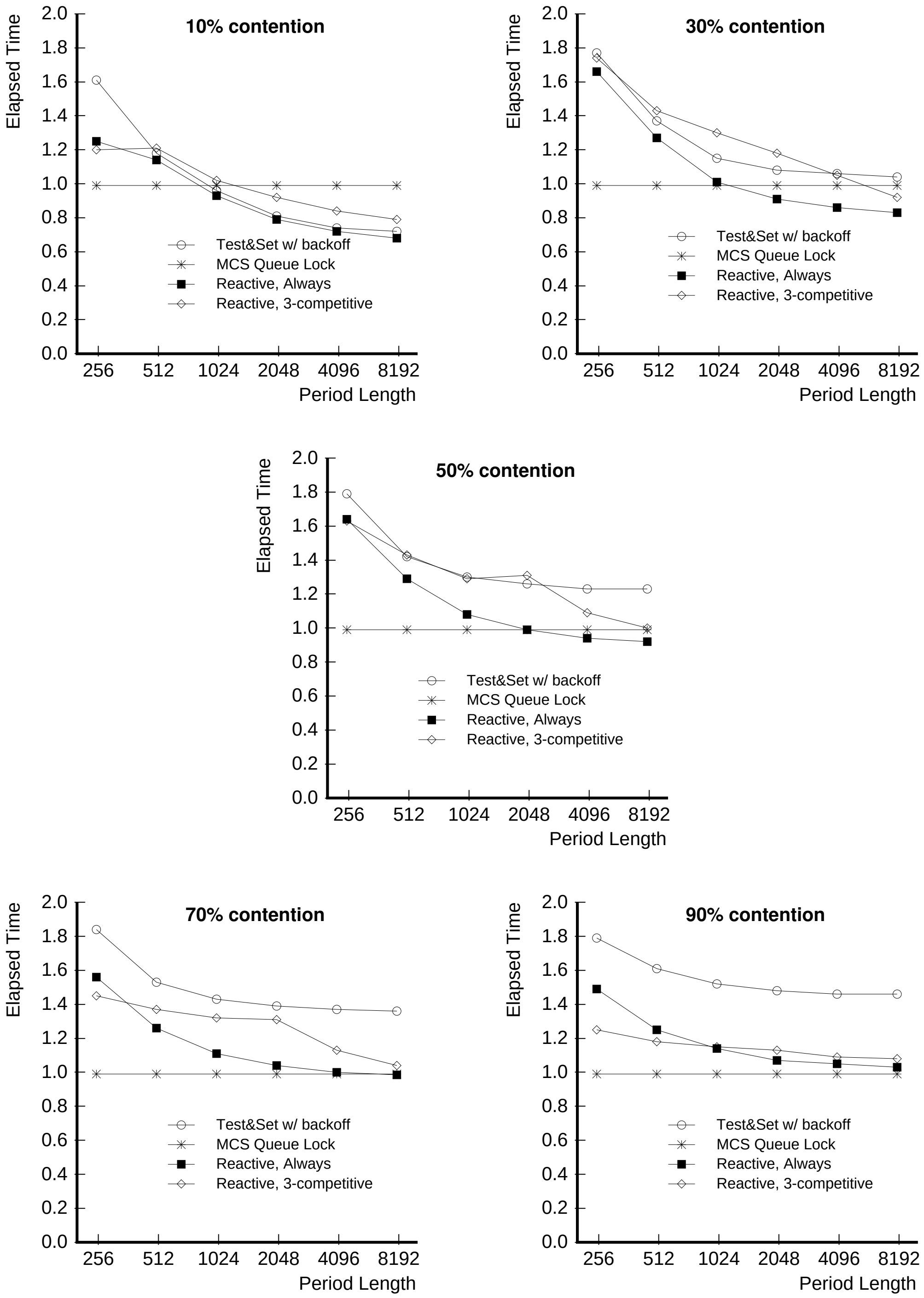


Figure 3.22: *Elapsed times for the time-varying contention test (normalized to the MCS Queue Lock) with a 3-competitive protocol-switching policy.* Period Length *is measured as the number of locks acquired per period.*

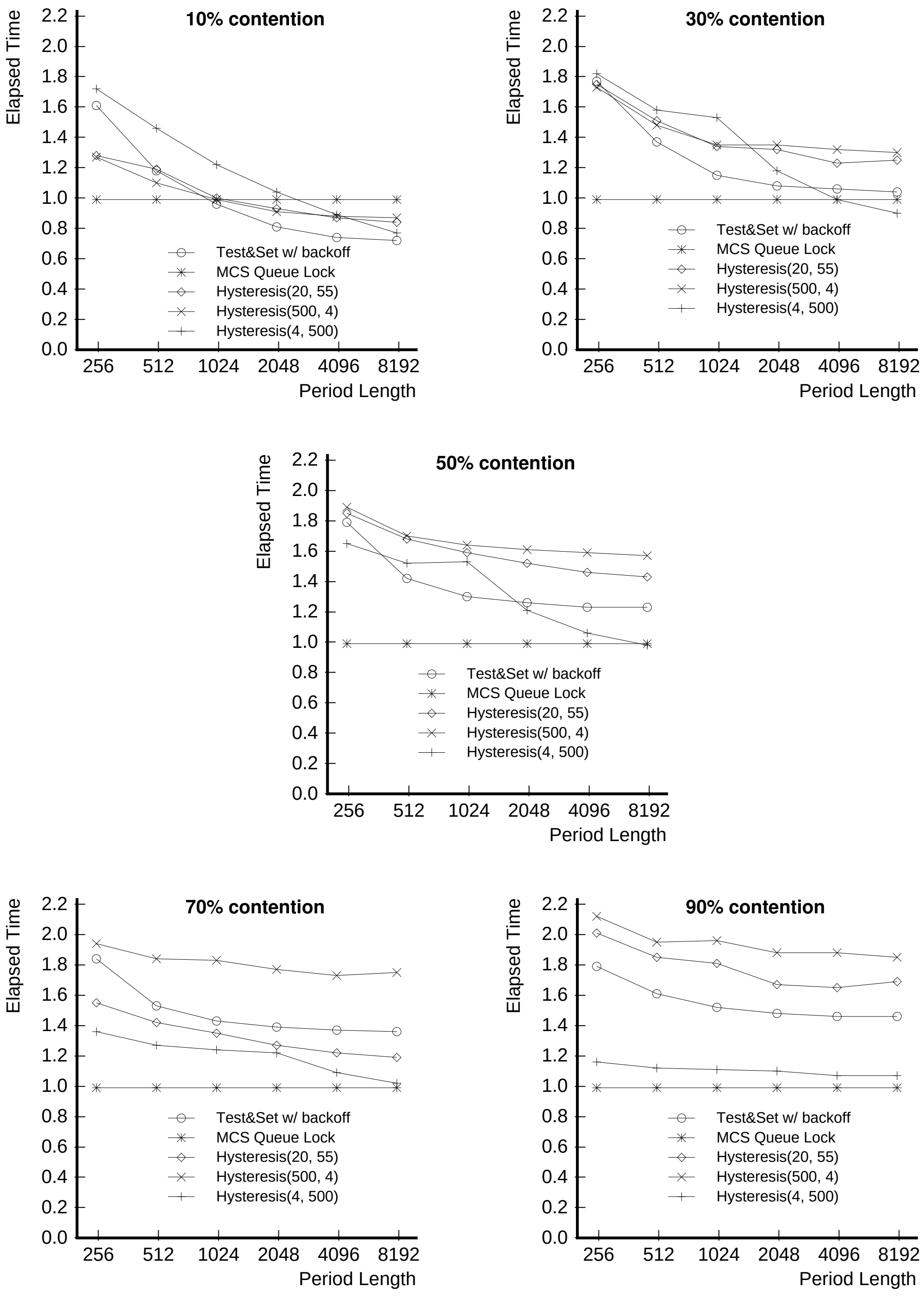


Figure 3.23: *Elapsed times for the time-varying contention test (normalized to the MCS Queue Lock) with a protocol-switching policy based on hysteresis.* Hysteresis(x, y) *switches from the test-and-test-and-set protocol to the MCS protocol after* x *consecutive high-contention lock requests, and switches from the the MCS protocol to the test-and-test-and-set protocol after* y *consecutive low-contention lock-requests.*

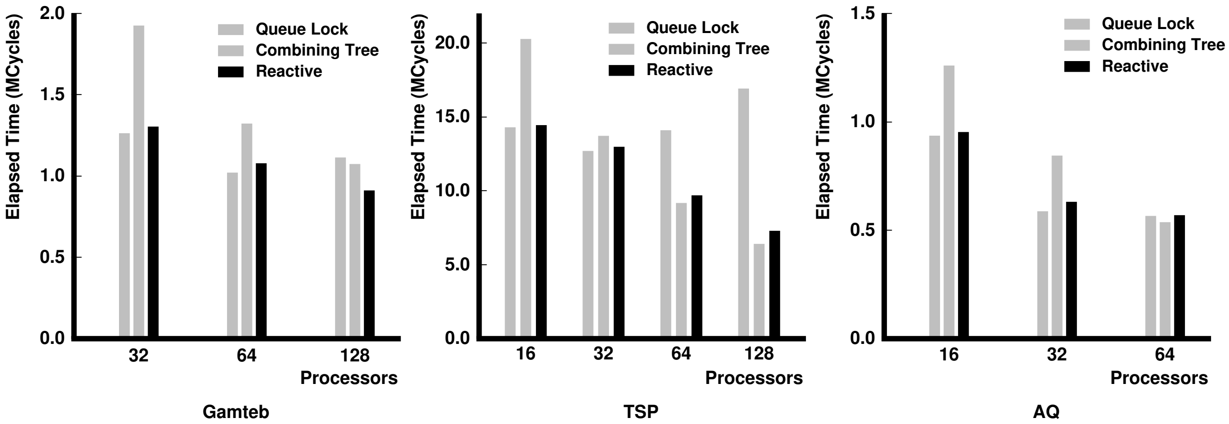


Figure 3.24: *Execution times for applications using different fetch-and-op algorithms.*

protocol. Among the hysteresis settings, we find that the best setting is Hysteresis(4, 500) that favors the MCS protocol under varying contention levels.

In summary, these results on alternative switching policies suggest that the 3-competitive algorithm is a good policy for deciding when to switch protocols. Moreover, its worst case performance is at most a constant factor worse that the performance of an optimal off-line algorithm. However, the price for guaranteeing this worst case bound is lower performance at intermediate switching frequencies.

These results also suggest that a policy that always switches protocols immediately after detecting that it is using a sub-optimal protocol works quite well, even though it has the potential for performing very poorly. To force the always-switch policy to thrash, contention levels would have to change rapidly. In practice however, contention levels cannot change arbitrarily rapidly. Once contention levels have built up, it takes a while for the contention to dissipate.

3.5.6 Application Performance

The synthetic benchmarks provide a precise characterization of the performance of the synchronization algorithms. We now investigate the impact of the synchronization algorithms on several parallel applications that use spin locks and fetch-and-op. The applications are written in C and parallelized with library calls. For each application, we vary the synchronization algorithm and measure the execution time on various numbers of processors.

Fetch-and-Op Figure 3.24 presents the execution times for applications that use fetch-and-op. We exclude the execution times for the test-and-test-and-set lock based fetch-and-op protocol: they are either slightly better or much worse than the execution times for the queue-based protocol. The combining trees are radix-2, and have as many leaves as the number of processors in each experiment.

Overall, the results show that the choice of fetch-and-op algorithms has a significant impact on the execution times, and that the reactive fetch-and-op algorithm selects the right protocol to execute in all cases. They demonstrate the utility of having a reactive algorithm select the protocol to use. To better understand the results, we describe the characteristics of each application

Gamteb Gamteb [11] is a photon transport simulation based on the Monte Carlo method. In this simulation, we used an input parameter of 2048 particles. Gamteb updates a set of nine interaction counters using fetch-and-increment.

On 32 and 64 processors, contention at all nine interaction counters are such that the queue-based protocol for fetch-and-op exhibits the best performance. The reactive algorithm selects the queue-based protocol for all the counters. On 128 processors, contention at one of the counters is high enough to warrant a combining tree. The reactive algorithm selects the combining tree protocol for that counter and the queue-based protocol for the other eight counters. This allows the reactive algorithm to outperform the passive algorithms that use the same protocol for all of the counters.

Traveling Salesman Problem (TSP) TSP solves the traveling salesman problem with a branch-and-bound algorithm. Processes extract partially explored tours from a global queue and expand them, possibly generating more partial tours and inserting them into the queue. In this simulation, TSP solves an 11-city tour. To ensure a deterministic amount of work, we seed the best path value with the optimal path. The global queue is based on an algorithm for a concurrent queue described in [18] that allows multiple processes simultaneous access to the queue. Fetch-and-increment operations synchronize access to the queue.

Contention for the fetch-and-increment operation in this application depends on the number of processors. With 16 and 32 processors, the queue-based fetch-and-op protocol is superior to the combining tree, but the opposite is true with 64 and 128 processors. The reactive algorithm selects the queue-based protocol at 16 and 32 processors, and the combining tree protocol at 64 and 128 processors.

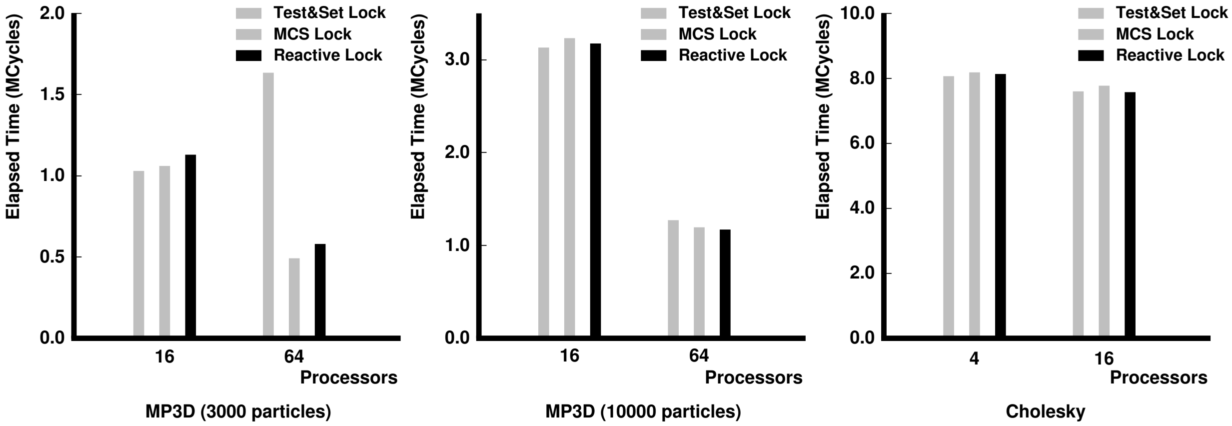


Figure 3.25: *Execution times for applications using different spin lock algorithms.*

Adaptive Quadrature (AQ) AQ performs numerical integration of a function with the adaptive quadrature algorithm. It proceeds by continually subdividing the range to be integrated into smaller ranges. A free processor dequeues a range to be integrated from a global queue. Depending on the behavior of the function in that range, the processor may subdivide the range into two halves, evaluating one half and inserting the other into the queue. In this simulation, AQ integrates the function $(sin(\pi i))^{30}$ in the range $(0, 30)$.

The queue implementation is the same as the one in TSP. However, computation grain sizes represented by each object in the parallel queue are larger compared to TSP, resulting in lower contention for the fetch-and-increment operation. At 16 and 32 processors, the queue-based fetch-and-op protocol is superior to the combining tree, but at 64 processors, both the queue-based and combining-tree protocols perform about equally. The reactive algorithm selects the queue-based protocol at 16 and 32 processors, and the combining tree protocol at 64 processors.

Spin Locks Figure 3.25 presents the execution times for applications with spin locks. Overall, the results show that while high contention levels might be a problem for the test-and-set lock, the higher latency of the MCS queue lock at low contention levels is not a significant factor. Computation grain sizes in between critical sections for these applications are large enough to render the higher latency of the queue lock insignificant. Thus, the reactive spin lock yields limited performance benefits over the MCS queue lock.

Nevertheless, the reactive spin lock achieves performance that is close to the best passive algorithm, and should be useful for applications that perform locking frequently and at a very fine grain such that lock latencies becomes a concern.

MP3D MP3D is part of the SPLASH parallel benchmark suite [52]. For this simulation, we use problem sizes of 3,000 and 10,000 particles and turn on the locking option in MP3D. We measure the time taken for 5 iterations. MP3D uses locks for atomic updating for cell parameters, where a cell represents a discretization of space. Contention at these locks is typically low. MP3D also uses a lock for atomic updating of collision counts at the end of each iteration. Depending on load balancing, contention at this lock can be high.

The higher latency of the MCS queue lock under low contention is not significant. On the other hand, the poor scalability of the test-and-set lock for updating collision counts significantly increases execution time for 3,000 particles on 64 processors. The reactive lock selects the test-and-test-and-set protocol for atomic updating of cell parameters, and selects the queue lock for updating collision counts.

Cholesky Cholesky is also part of the SPLASH parallel benchmark suite. It performs Cholesky factorization of sparse, positive definite matrices. Due to speed and space limitations of the Alewife simulator, we could only factorize small matrices with limited amounts of parallelism. In this simulation, we factorize an 866x866 matrix with 3,189 non-zero elements. We do not intend this simulation to be indicative of the behavior of the SPLASH benchmark, but rather as a test for comparing the spin lock algorithms. As in MP3D, we see that the higher latency of the MCS lock has a negligible impact on execution times.

3.6 Reactive Algorithms and Message-Passing Protocols

In this section, we consider reactive algorithms that select between shared-memory and message-passing protocols. Recent architectures for scalable shared-memory multiprocessors [31, 34, 48] implement the shared-memory abstraction on top of a collection of processing nodes that communicate via messages through an interconnection network. They allow software to bypass the shared-memory abstraction and directly access the message layer . This provides an opportunity for software to use message-passing protocols to implement synchronization operations.

The advantage of using message-passing to implement synchronization operations over shared-memory is that under high contention, message-passing results in more efficient communication patterns, and atomicity is easily provided by making message handlers atomic with respect to other message handlers [54]. For example, fetch-and-op can be implemented by allocating the fetch-and-op variable in a private memory location of some processing node. To perform a fetch-and-op, a process sends a message to the processor associated with that memory location. The message handler computes the operation using that memory location and returns the result with its reply. This results in the theoretical minimum of two messages to perform a fetch-and-op: a request and a reply. Contrast this with shared-memory protocols for fetch-and-op that require multiple messages to ensure atomic updating of the fetch-and-op variable.

With more efficient communication patterns and atomic message handling, message-passing protocols can outperform corresponding shared-memory protocols when contention is high. On the other hand, the fixed overheads of message sends and receives make message-passing protocols more expensive than corresponding shared-memory protocols when contention is low. This diminishes the advantage of message-passing protocols unless the level of contention can be predicted. Once again, we have a contention-dependent choice to make between protocols. Fortunately, reactive algorithms will allow a run-time choice between shared-memory and message-passing protocols.

Using the framework based on consensus objects, we designed reactive algorithms for spin locks and fetch-and-op that select between shared-memory and message-passing protocols. Unlike the shared-memory protocols that reach the in-consensus phase by acquiring and releasing locks to access a consensus object, the message-passing protocols reach the in-consensus phase as part of an atomic message handler. For example, in a message-passing based combining tree, a message ultimately gets sent to the root of the combining tree. The message handler for the root represents a process that is in the in-consensus phase.

The reactive spin-lock algorithm selects between a test-and-test-and-set lock protocol and a message-passing queue lock protocol. The message-passing queue lock is implemented by designating a processor as a lock manager. To request a lock, a process sends a message to the lock manager and waits for a reply granting it the lock. The lock manager maintains a queue of lock requesters and responds to lock request and reply messages in the obvious way.

The reactive fetch-and-op algorithm selects between a test-and-test-and-set lock based protocol, a centralized message-passing fetch-and-op protocol (described above), and a message-passing combining-tree protocol. The message-passing combining tree protocol uses messages to traverse the combining tree. To execute a fetch-and-op, a process sends a message to a leaf of the tree. After polling the network to detect messages to combine with, a message handler relays a message to its parent. In this way, messages combine and propagate up to the root of the combining tree where the operation is performed on the fetch-and-op variable.

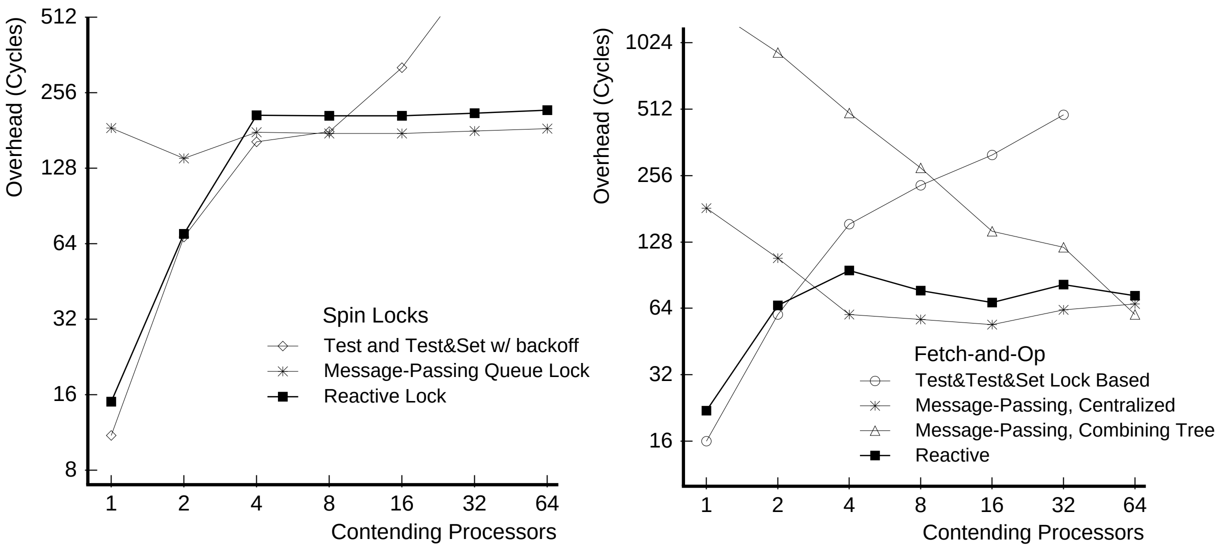


Figure 3.26: *Baseline performance comparing shared-memory and message-passing protocols for spin locks and fetch-and-op. The reactive algorithms select between the shared-memory and message-passing protocols.*

Figure 3.26 presents the baseline performance of the reactive algorithms that select between shared-memory and message-passing protocols. Like the reactive algorithms that select between purely shared-memory protocols, these reactive algorithms also succeed in selecting the right protocol for a given level of contention. As a demonstration of the advantage of using message-passing over shared-memory protocols, note that under high contention, the message-passing fetch-and-op protocols result in lower overhead and correspondingly higher throughput than the shared-memory fetch-and-op protocols. The numbers also show that on Alewife, the message-passing queue lock is always inferior to the shared-memory MCS queue lock. However, on architectures with different communication overheads and levels of support for shared-memory, the reverse might be true.

3.7 Implementing a Protocol Selection Algorithm

This section overviews the steps involved in the practical implementation of a protocol selection algorithm, and presents pseudo-code for the reactive spin lock as a concrete example. This section is intended to guide the reader in implementing his/her own protocol selection algorithms. It is hard to quantify the effort necessary for implementing new protocol selection algorithms. As an example of the level of effort, it took about a week to profile the component protocols and to implement and tune the performance of each of the reactive spin lock and fetch-and-op algorithms.

The design and implementation process proceeds in two phases. In the first phase, we obtain a correct implementation of a protocol selection algorithm, using some dummy policy for selecting protocols. We will describe a possible structure for the implementation by subdividing it into four distinct parts. In the second phase, we implement the policy module for selecting protocols and tune the policy for a given machine architecture.

3.7.1 Phase 1: Implementing a Correct Algorithm

The implementation process is largely based on the framework presented in Section 3.2 for designing protocol selection algorithms. We are given a set of pre-existing protocols that exhibit a tradeoff that depends on some run-time condition, and the task is to design and implement a correct algorithm for selecting among them. We begin by identifying the consensus object in each of the protocols. If a protocol does not have a consensus object and does not satisfy the properties associated with consensus objects described in Section 3.2, then some other method must be found for serializing protocol changes at the protocol. Once the consensus objects have been identified, we can proceed to implement the protocol selection algorithm.

For descriptive purposes, it will be convenient to organize the implementation code in four parts:

1. The data structures.
2. The dispatch procedure.
3. The protocols.
4. The protocol change procedures.

The data structures

The data structures are composed of the original data structures of each component protocol, and a mode variable. For example:

```
type reactive_data = record
    mode  : (PROT1, PROT2)
    prot1 : protocol1_data
    prot2 : protocol2_data
```

There may also need to be other slots for storing run-time statistics for use by the policy module.

On a cache-coherent architecture, it may be necessary to place the mode slot of the record in a separate cache line due to false-sharing concerns. We expect the mode variable to be mostly read-only, and this may conflict with a frequently written portion of the protocol data structures. An alternative approach is to use pointers to the original protocol data structures so that the entire `reactive_data` record is mostly read-only.

The dispatch procedure

The dispatch procedure uses the mode variable to dispatch to one of the protocols in use:

```
procedure dispatch(r : ^reactive_data) returns V
    case r->mode
        PROT1: return run_prot1(r->prot1)
        PROT2: return run_prot2(r->prot2)
```

Note that it is possible for the mode variable to change in between when it is read to when the protocol is run. Therefore the mode variable exists only as a hint to expedite the dispatch. We rely on the consensus object to detect invalid protocol executions.

In some cases, it may be possible to optimistically execute a protocol without checking the mode variable in order to optimize for latency in the absence of contention. The reactive spin lock pseudo-code presented below uses this optimization.

The protocols

The protocols need to be modified to monitor the run-time conditions that determine the tradeoff among the protocols. This monitoring code is protocol specific. The pseudo-code for the reactive spin lock presented below provides an example of how to monitor the level of contention. We found that a small level of hysteresis is necessary to obtain a reliable estimate of run-time conditions. For example, the reactive spin lock waits until 4 consecutive lock acquisitions find an empty queue before indicating to the policy module that the test-and-test-and-set protocol should be used.

To minimize the impact on latency, one should avoid placing the monitoring code in a critical paths of a protocol should be inserted. An ideal place to insert monitoring code is in busy-wait loops.

We modify the in-consensus code of each protocol to check the mode variable to see if the protocol is valid. If not, the process will have to abort and retry the synchronization operation, taking care to signal any other processes that are waiting on it to abort also. After aborting, the protocol can call the dispatch procedure directly instead of returning to it.

We also add some code to the in-consensus phase of each protocol to decide if a protocol change is necessary, and to call the corresponding protocol change procedures. At this stage of the implementation process, we can defer the task of implementing a good policy by inserting a dummy stub to change protocols. In fact, it is a useful debugging aid to insert a stub that randomly requests protocol changes. This allowed us to exercise the protocol change procedures and exposed a number of errors in our initial implementations.

The protocol change procedures

As described in Section 3.2, changing protocols with consensus objects is straightforward. As an optimization, the reactive algorithm guarantees to call a protocol change procedure only from the in-consensus phase of a valid protocol execution. Therefore, a protocol change procedure only needs to acquire the consensus object of the protocol to change to: the consensus object of the current protocol has already been acquired. The protocol change procedure needs to update the new protocol, update the mode variable to point to the new protocol, and may also need to signal any processes left waiting in the old protocol.

3.7.2 Phase 2: Policy and Performance Tuning

The next phase concerns implementing the policy for deciding when to change protocols. Rather than being concerned with correctness, this phase concentrates on performance tuning.

In order to determine the tradeoffs and the breakeven points between the protocols, the implementor has to run a set of tests akin to the baseline test presented in Section 3.5. Unfortunately, these tradeoffs are architecture dependent and thus have to be measured for each target machine architecture. We expect this to be the most time-consuming part of the implementation process.

Based on these tradeoffs and the run-time statistics collected by the monitoring code, the policy module can decide whether the current protocol is optimal or not. Here, we have a choice between a straightforward, always-switch policy and the 3-competitive policy described in Section 3.4.

As a concrete example of the implementation process, let us examine the implementation of a reactive spin lock.

```
type release_mode = (TTS, TTS_TO_QUEUE, QUEUE, QUEUE_TO_TTS)

type qnode = record
    next   : ^qnode
    status : (WAITING, GO, INVALID)
    empty_queue : int

// The mode slot should reside in a different cache line from the other slots
// Initial values are either {FREE, INVALID, TTS} or {BUSY, nil, QUEUE}
type lock = record
    mode       : (TTS, QUEUE)            // mode variable
    tts_lock   : (FREE, BUSY)            // slot for TTS lock
    queue_tail : (INVALID, ^qnode)       // slot for queue lock

procedure acquire_lock (L : ^lock, I : ^qnode) returns release_mode
    if test_and_set (&L->tts_lock) = FREE // optimistically try TTS lock
        return TTS
    else if L->mode = TTS
        return acquire_tts (L, I)        // try TTS lock
    else
        return acquire_queue (L, I)      // try queue lock

procedure release_lock (L : ^lock, I : ^qnode, mode : release_mode)
    case mode of
        TTS:          release_tts(L)                // release TTS lock
        QUEUE:        release_queue (L, I)          // release queue lock
        TTS_TO_QUEUE: release_tts_to_queue (L, I)   // change to QUEUE mode
        QUEUE_TO_TTS: release_queue_to_tts (L, I)   // change to TTS mode
```

Figure 3.27: *Reactive spin lock: data structures and top-level dispatch code.*

3.7.3 The Reactive Spin Lock

Figures 3.27–3.29 present the pseudo-code for our reactive spin lock. The reactive spin lock selects between the test-and-test-and-set protocol and the MCS queue lock protocol. Figure 3.27 presents the data structures and the top-level dispatch procedure. The data structure is composed of the test-and-test-and-set lock and the MCS queue lock data structures, and a mode variable. The mode variable indicates which of the two locks is valid. `acquire_lock` and `release_lock` acquire and release the reactive lock, respectively.

The dispatch procedure, `acquire_lock`, attempts to acquire the reactive lock by checking the mode variable to decide which protocol to use. `acquire_lock` returns a value that informs `release_lock` which protocol to use in releasing a lock and whether a mode change is requested. Alternatively, this information can be communicated through the mode variable or some other shared variable.

```
    procedure acquire_tts (L : ^lock, I: ^qnode) returns release_mode
>       mode : release_mode := TTS
M>      retries : integer := 0                // count of number of failed attempts
        repeat while TRUE
            if L->tts_lock = FREE
                if test_and_set (&L->tts_lock) = FREE
>                   return mode
M,P>            if retries++ > TTS_RETRY_LIMIT
>                   mode := TTS_TO_QUEUE      // change to QUEUE mode upon release
            delay ()                          // do backoff
>           if L->mode != TTS
>               return acquire_queue (L, I)   // mode changed to QUEUE

    procedure release_tts (L : ^lock)
        L->tts_lock := FREE

    procedure acquire_queue (L : ^lock, I: ^qnode) returns release_mode
        I->next := nil
        predecessor : ^qnode := fetch_and_store (&L->queue_tail, I)
        if predecessor = nil                  // queue was empty, lock acquired
P>          if I->empty_queue++ > EMPTY_QUEUE_LIMIT  // switch mode?
>               return QUEUE_TO_TTS
>           else
>               return QUEUE
>       else if predecessor != INVALID        // queue was non-empty
            I->status := WAITING
            predecessor->next := I
M>          I->empty_queue := 0
            repeat while I->status = WAITING  // wait for GO or INVALID signal
>           if I->status = GO                 // lock acquired
>               return QUEUE
>           else                              // queue was invalid
>               return acquire_tts (L, I)
>       else                                  // queue was invalid
>           invalidate_queue (L, I)           // invalidate others on the queue
>           return acquire_tts (L, I)

    procedure release_queue (L : ^lock, I : ^qnode)
        if I->next = nil                          // no known successor
            old_tail : ^qnode := fetch_and_store (&L->queue_tail, nil)
            if old_tail = I return                // I really had no successor
            usurper : ^qnode := fetch_and_store (&L->queue_tail, old_tail)
            repeat while I->next = nil
            if usurper != nil
                usurper->next := I->next; return
        I->next->status := GO
```

Figure 3.28: *Reactive spin lock: component protocols. Modifications to the original protocols are marked by "`>`". "`M>`" denotes modifications for monitoring run-time conditions, while "`P>`" denotes modifications for implementing the policy for changing protocols.*

```
procedure release_tts_to_queue (L, I)
    acquire_invalid_queue (L, I)
    L->mode := QUEUE
    release_queue (L, I)

procedure release_queue_to_tts (L, I)
    L->mode := TTS
    invalidate_queue (L, I)
    release_tts (L)

procedure acquire_invalid_queue (L : ^lock, I: ^qnode)
    // L->queue_tail should be INVALID or point to tail of an invalid queue
    repeat while TRUE
        I->next := nil
        predecessor : ^qnode := fetch_and_store (&L->queue_tail, I)
        if predecessor = INVALID return
        // got on to tail of an invalid queue, wait for INVALID signal and retry
        I->status := WAITING
        predecessor->next := I
        repeat while I->status = WAITING

procedure invalidate_queue (L : ^lock, head : ^qnode)
    tail : ^qnode := fetch_and_store (&L->queue_tail, INVALID)
    repeat while head != tail
        repeat while head->next = nil
        next : ^qnode := head->next
        head->status := INVALID
        head := next
    head->status := INVALID
```

Figure 3.29: *Reactive spin lock: making protocol changes. These routines are called only by processes that have successfully acquired one of the component locks.* `invalidate_queue` *can only be called by a process that has acquired a queue lock, either in a valid or invalid state.*

To optimize for latency in the absence of contention, `acquire_lock` avoids checking the mode variable by optimistically attempting to acquire the test-and-test-and-set lock. The mode variable is checked only if the attempt fails. This potentially increases the amount of work in acquiring a lock when contention is high and the lock is in queue mode. However, if we place both sub-locks in the same cache line, the optimistic *test&set* attempt will pre-fetch the queue lock and avoid any further bus or network transactions when attempting to acquire the queue lock. Furthermore, the optimistic attempt uses processor cycles that would have been otherwise unproductively spent spin waiting.

Figure 3.28 presents the pseudo-code of the protocols being selected. The original protocols have been modified to detect mode changes, and to abort and retry the synchronization operation upon detecting that the protocol is invalid. `acquire_tts` and `release_tts` implement the test-and-test-and-set protocol while `acquire_queue` and `release_queue` implement the MCS queue lock protocol. Modifications to the original protocols are marked by "`>`" on the left end of each line. Additionally, "`M>`" denotes modifications for monitoring run-time conditions, while "`P>`" denotes modifications for implementing the policy for changing protocols.

The original protocols have also been modified to monitor run-time conditions, as described in Section 3.3. `TTS_RETRY_LIMIT` and `QUEUE_EMPTY_LIMIT` are parameters that control when the reactive algorithm decides to switch modes. If a process fails to acquire a test-and-test-and-set lock after `TTS_RETRY_LIMIT` *test&set* attempts, it will change to `QUEUE` mode the next time it acquires the lock. If a process detects an empty queue during `QUEUE_EMPTY_LIMIT` consecutive lock acquisitions, it will change to `TTS` mode upon the next lock release.

Finally, Figure 3.29 presents the pseudo-code for performing the mode changes. `release_tts_to_queue` changes from the test-and-test-and-set protocol to the queue lock protocol and `release_queue_to_tts` changes from the queue lock protocol to the test-and-test-and-set protocol. In order to ensure that protocol changes are serializable with respect to other protocol executions and changes, these procedures are called only by processes that have acquired a valid consensus object

3.8 Summary

This chapter explores the design and performance implications of dynamic protocol selection. It demonstrates how a reactive algorithm that dynamically selects synchronization protocols in response to run-time conditions can outperform a passive algorithm that uses a fixed protocol.

We identified two main challenges in designing a protocol selection algorithm. The first is in designing an efficient method for selecting and changing protocols, and the second is in providing intelligent policies for deciding when to change protocols. This chapter describes a framework for designing and reasoning about protocol selection algorithms. It introduces the notion of consensus objects that allows a reactive algorithm to select and change protocols correctly and efficiently. Consensus objects allow a synchronizing process to optimistically execute a protocol without prior coordination with other synchronizing processes. This chapter also describes several policies for changing protocols and presents a 3-competitive policy.

Accordingly, the implementation of a protocol selection algorithm should proceed in two parts. In the first part, the algorithm should be designed so that protocols can be selected and changed efficiently. If the design framework presented in this chapter is applicable, then designing this part should be straightforward. In the second part, the tradeoffs among the protocols need to be measured so that the policy for changing protocols can be implemented and tuned.

To demonstrate the effectiveness of dynamic protocol selection, we designed and implemented reactive spin-lock and fetch-and-op algorithms, and compared their performance against the best passive algorithms on the Alewife multiprocessor. The performance results show that reactive algorithms succeed in achieving performance that is close to the best static choice of protocols, and that they do so with minimal run-time overhead. These results suggest that run-time adaptation is an effective way for reducing synchronization costs in a parallel program. This has the important advantage of relieving the programmer from the difficult task of selecting the best protocol for synchronization operations.

Chapter 4

Waiting Algorithms

The previous chapter shows how a protocol selection algorithm can dynamically choose among several protocols to implement a synchronization operation. By tailoring the protocol to the level of contention experienced at run-time, a reactive algorithm achieves efficient and robust performance.

In this chapter, we focus on waiting algorithms that dynamically choose among waiting mechanisms to wait for synchronization. Waiting algorithms can reduce the cost of waiting by overlapping waiting time with other computation. This is achieved by invoking a signaling mechanism that switches processor execution to another runnable thread. However, since a signaling mechanism incurs a significant fixed cost, we have to be careful about when to invoke it. Thus, a waiting algorithm has to make a run-time dependent choice among waiting mechanisms.

Unlike dynamically changing protocols, dynamically changing waiting mechanisms is a local operation that does not need to be coordinated among other participating processes. Thus, providing an efficient run-time method for changing waiting mechanisms does not present a problem. The challenge is in designing intelligent policies for deciding when to switch waiting mechanisms.

In this chapter we design and analyze two-phase waiting algorithms that choose between polling and signaling waiting mechanisms. Recall that a waiting thread first polls until the cost of polling reaches a limit L_{poll}. If further waiting is necessary, the thread resorts to a signaling mechanism. We first describe several common waiting mechanisms and their associated costs. We then model the problem of choosing between waiting mechanisms as a task system, and show how constraints on the inputs of the task system allow us to improve the competitive factors of waiting algorithms. To this end, we introduce the notion of a *restricted adversary* that is constrained to choose waiting times from a predetermined probability distribution.

We use competitive analysis to help design the waiting algorithms. We first develop a probabilistic model of the expected costs of waiting algorithms. We then develop models of waiting time distributions for several common synchronization types. The cost model, together with the waiting time distributions, allows us to design waiting algorithms that approach optimal performance against restricted adversaries.

We manage to improve upon the competitive factors of two-phase waiting algorithms while minimizing the run-time cost of making the decision. In particular, we are able to prescribe *static* choices of L_{poll} for a two-phase waiting algorithm such that the resulting waiting algorithm achieves close to the optimal on-line competitive factor of 1.58 against a restricted adversary.

To corroborate the theoretical analysis, we present experimental results that measure the distribution of waiting times and the performance of two-phase waiting algorithms in several parallel applications. The results show that two-phase waiting is indeed a robust waiting algorithm and achieves performance close to the best static choice of waiting mechanisms.

4.1 Waiting Mechanisms

Before we can model the cost of waiting, we need to model the costs of the waiting mechanisms that are available to a waiting algorithm. We describe here the implementation and the waiting costs of *spinning, blocking, switch-spinning, and switch-blocking*. Spinning and blocking are the most common waiting mechanisms used in multiprocessing environments. Switch-spinning and switch-blocking are additional waiting mechanisms that multithreaded multiprocessors, such as Alewife, may provide. We model the waiting costs as a function of t, the waiting time.

Spinning A thread spin-waits by periodically reading the value of a memory location. In cache-coherent multiprocessors the memory location is cached locally to avoid network traffic while spinning. A change to the state of the memory location due to a write is communicated to the waiting threads through the ensuing cache invalidations. Because spin-waiting cycles are wasted, the waiting cost of spinning for t cycles is simply equal to the waiting time, t.

Blocking Blocking a thread involves unloading it, and at a later time, reenabling and reloading it. Thus, a blocked thread allows other threads to use the processor. Blocking

	Action	Instructions	Base Cycles
Unloading	Unload registers	21 stores	63
	Enqueue thread	2 stores 2 loads 7 other	17
	Book-keeping	6 stores 1 load 6 other	26
Reenabling	Lock queue of blocked threads	2 loads 1 store 6 other	13
	Queue on processor ready queue	6 loads 5 stores 12 other	39
Reloading	Reload registers	21 loads	42
	Restore misc. state	1 load 6 other	8
	Book-keeping	1 store 8 other	11
	Total	114	219

Table 4.1: *Breakdown of the cost of blocking in Alewife.*

incurs a fixed cost, B, that depends on the number of processor cycles needed to perform the necessary thread scheduling and descheduling.

On Alewife, a blocked thread is placed on a software queue associated with the failed synchronization. When signaled to proceed, the thread is reenabled, and eventually rescheduled and reloaded. In the experiments, the cost of blocking on Alewife is approximately 500 cycles.

Table 4.1 gives a breakdown of the costs of unloading, reenabling, and reloading a thread in terms of instructions and base-cycle times in Alewife (base cycles assume cache hits). In terms of base cycles, the cost of blocking is 219 cycles. However, the measured cost of blocking is experimentally observed to be about 500 cycles because of cache misses. Of the measured cycles, about 300 cycles are spent unloading the task, 100 cycles reenabling it and 65 cycles reloading it. Loads and stores are observed to take 3 times longer than the base-cycle time when unloading a thread due to cache misses. Since an unloaded thread usually resides in the cache, reloading a thread takes close to the base-cycle time.

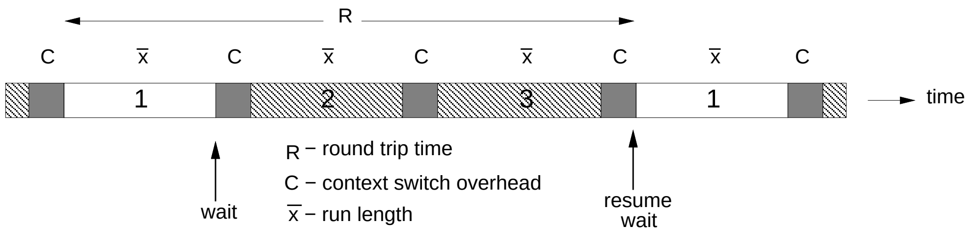


Figure 4.1: *Switch-Spinning – time line of three active contexts sharing a processor. A switch-spinning thread occupies context 1 and its waiting time is interleaved with executions of threads in context 2 and context 3.*

Switch-Spinning On a multithreaded processor, a waiting thread can switch rapidly to another processor-resident thread in a round-robin fashion, allowing the waiting time to be overlapped with useful computation by other threads. Control eventually returns to the waiting thread and the synchronization variable is re-polled. Switch-spinning is therefore a polling mechanism. Since other threads are allowed to utilize the processor, this is a more efficient polling mechanism than spinning.

We model the cost of switch-spinning for t cycles as t/β, where β represents the relative efficiency of switch-spinning over spinning. In other words, a switch-spinning thread that waits for t cycles wastes only t/β processor cycles. The following analysis models the value of β in a block-multithreaded processor.

Figure 4.1 illustrates a switch-spinning scenario with three hardware contexts. β depends on the number of hardware contexts, N, the context switch overhead, C, and the run length. Let $\bar{x}$ be the mean run length. Run length is the time between the instant a thread starts executing on the processor to the instant it encounters a context switch. Let R be the round-trip time, defined as the time between successive context switches to the same switch-spinning thread. Thus, $R \approx N(\bar{x} + C)$.

Suppose that a thread has to wait for t cycles. Control will return to the waiting thread $\lceil \frac{t}{R} \rceil$ times before it can proceed. To simplify the analysis, assume that a switch-spinning thread also has a mean run-length of $\bar{x}$ so that the cost of waiting is increased by $\bar{x} + C$ cycles each time control returns to the waiting thread. Therefore, the waiting cost of switch-spinning for t cycles is approximately $\lceil \frac{t}{R} \rceil (\bar{x} + C)$ cycles. On Alewife, $N = 4$ and $C = 14$ cycles.

We now approximate β. If t is shorter than R, then $\beta = t/C$. Hence, in this case, switch-spinning is more efficient than spinning if $t > C$. If t is long compared to R, we can ignore the ceiling operator and obtain $\beta = N$. This is commonly the case in our simulations. Thus switch-spinning amortizes the cost of polling among the N contexts.

Switch-Blocking Switch-blocking is a mechanism where a waiting thread *disables* the hardware context in which it is resident, in addition to switching to another processor-resident thread. As in blocking, the waiting thread is placed on a queue associated with the failed synchronization. Further context switches bypass the disabled context until it is reenabled. Since there is no need to load and unload threads, switch-blocking is a signaling mechanism with lower fixed cost than blocking.

We estimate the cost of switch-blocking in Alewife to be less than 100 cycles. We do not analyze the performance of switch-blocking as a waiting mechanism in this thesis. In [16], Gopinath *et al.* present an analysis of switch-blocking on Alewife that shows that the use of switch-blocking as a waiting mechanism does not yield much advantage over switch-spinning, given the current parameters of the Alewife machine.

4.2 Polling versus Signaling

A waiting algorithm can use any of the above waiting mechanisms to wait for synchronization. Due to significant differences between the waiting costs of waiting mechanisms, this choice can be critical to performance. However, it is hard to make a correct choice without knowledge of waiting times. Long waiting times hurt the performance of spinning and switch-spinning. On the other hand, blocking incurs a significant fixed cost because of the need to deschedule and reschedule waiting threads, and the need to save and restore processor state.

It turns out that the fundamental choice of waiting mechanisms is between polling and signaling mechanisms. Spinning and switch-spinning are examples of polling mechanisms, while blocking and switch-blocking are examples of signaling mechanisms. We can model the cost of any polling mechanism as proportional to waiting time, and the cost of any signaling mechanism as a fixed constant, independent of waiting time. Thus, in our analysis, we denote the cost of polling for t cycles as t/β, and the cost of signaling as a fixed cost B.

4.2.1 Polling vs. Signaling as a Task System

Just as for the problem of choosing between protocols, we can model the problem of choosing between a polling and signaling mechanism as a task system. Figure 4.2 illustrates such a task system. Each of the two states represents each of the waiting mechanisms. A request sequence for this task system is composed of two types of tasks: *wait* and *proceed*.

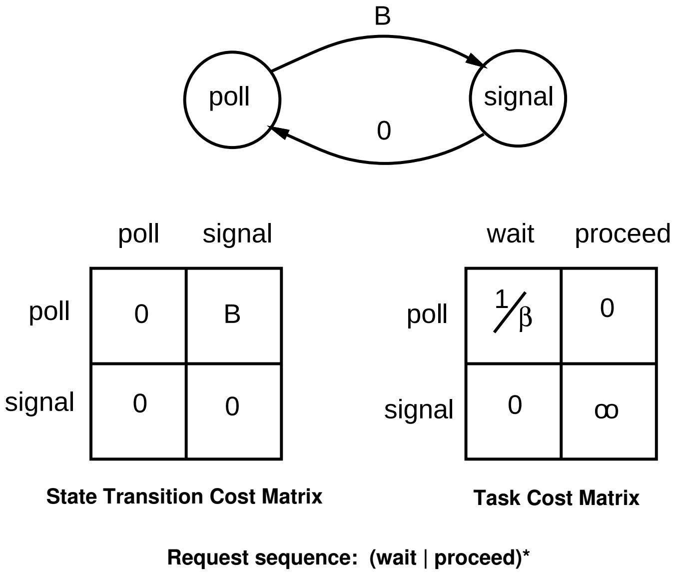


Figure 4.2: *A task system that offers a choice between polling and signaling mechanisms.*

For each input request, an on-line algorithm has to choose which state to process the request. It is allowed to perform the state transition before servicing the request.

A synchronization wait of t cycles presents the task system with a sequence of t wait requests followed by a proceed request. Thus, a request sequence is composed of contiguous sequences of wait requests, each followed by a proceed request, and can be described by the regular expression $(\texttt{wait}|\texttt{proceed})^*$.

The work by Borodin, Linial and Saks [9] presents a 3-competitive algorithm for this type of task system (See Chapter 2). Their work also shows that the competitive factor of 3 is a lower bound for a general two-state task system with unconstrained inputs. Fortunately, the special structure of this task system allows on-line waiting algorithms to achieve smaller competitive factors.

The task system's initial state is the polling state. To model the fact that each synchronization wait is processed starting from the polling state, the task system must return to the polling state at the end of each synchronization wait. To achieve this, we designate the cost of processing a proceed request in the signaling state as infinite. Thus, any reasonable algorithm returns to the polling state after each synchronization wait has been processed.

It is well known that polling until the cost of polling equals the cost of signaling yields a 2-competitive algorithm. In Section 4.4, we explore how constraints on the input request sequences allow us to achieve even smaller competitive factors. We place constraints on the distribution of waiting times, thus constraining the run-lengths of wait requests to be selected from a probability distribution.

4.3 Two-Phase Waiting Algorithms

An on-line algorithm that chooses between polling and signaling mechanisms is the two-phase waiting algorithm, first suggested by Ousterhout in [47]. In a two-phase waiting algorithm a waiting thread first uses a polling mechanism to wait until the cost of polling reaches a limit L_{poll}. If further waiting is necessary at the end of the polling phase, the thread resorts to a signaling mechanism, incurring a fixed cost B.

The choice of L_{poll} determines the performance of two-phase waiting algorithms. In a sense, two-phase waiting is a generalization of the always-spin and always-block algorithms: it introduces a continuum of choices between always-block ($L_{poll} = 0$) and always-spin ($L_{poll} = \infty$).

Thus, we transform the problem of deciding between polling and signaling into the problem of deciding the value of L_{poll} in a two-phase algorithm. In other words, the task of a waiting algorithm is to decide how long to poll before resorting to a signaling mechanism. Because of the need to minimize run-time overhead, the method explored in this thesis is to choose L_{poll} statically, based on knowledge of likely waiting-time distributions for different synchronization types.

4.3.1 Static Two-Phase Waiting Algorithms

The choice of L_{poll} can either be made statically at compile time, or dynamically at run-time. In [26], Karlin *et al.* present randomized and adaptive methods for dynamically determining L_{poll}. A drawback of these methods is that they incur a significant run-time overhead. Minimizing the run-time overhead for determining L_{poll} is crucial in large-scale multiprocessors that support lightweight threads. In such systems, the cost of signaling mechanisms can be as small as a few hundreds of cycles.

In this thesis, we focus on *static* methods for determining L_{poll} so as to minimize the run-time overhead of choosing waiting mechanisms. Our approach exploits the randomization inherent in the waiting times encountered in synchronization to improve the competitive factors and achieve robust performance. In practice, we expect each synchronization type to exhibit waiting times that are randomly distributed according to some waiting-time distribution. For example, waiting times for producer-consumer synchronization are exponentially distributed under Poisson arrivals of synchronizing threads.

The next section will provide a framework for analyzing the relative performance of different static choices of L_{poll} under different waiting time distributions, and show that we can use readily available knowledge of synchronization types and their characteristic waiting time distributions to guide our choice of L_{poll}.

4.4 Analysis of Waiting Algorithms

We transform the problem of choosing between waiting mechanisms to one of choosing the right value of L_{poll} for a two-phase waiting algorithm. In this section, we model the expected waiting costs of various waiting algorithms and derive optimal values for L_{poll}. Our analysis assumes that we can always find a runnable thread to replace a blocked thread. We compare the performance of the following algorithms:

poll	– always-poll.
signal	– always-signal.
2phase/α	– two-phase waiting with $L_{poll} = \alpha B$.
Opt	– optimal off-line.

The analysis proceeds as follows. We first describe the notion of adversaries and observe how weaker adversaries allow on-line waiting algorithms to choose L_{poll} so as to achieve better competitive factors. We then model the expected waiting cost of two-phase waiting algorithms as a function of waiting time distributions and of the constituent waiting mechanisms. We consider several common synchronization types and show how they naturally lead to exponential and uniformly distributed waiting times.

From the cost model and the waiting time distributions, we derive values for L_{poll} such that static two-phase waiting algorithms can achieve close to optimal on-line competitive factors. In particular, we derive optimal values for L_{poll} under exponential and uniform waiting-time distributions. Figure 4.3 illustrates the flow of the analysis. We summarize the results of the analysis in Section 4.5.3.

4.4.1 Competitive Algorithms and Adversaries

As the preceding discussion on task systems observes, on-line waiting algorithms can achieve competitive factors that are smaller than 3 because the pattern of input request sequences has to satisfy some constraints. It is useful to model such constraints as restrictions that are placed on an *adversary*, as explained below.

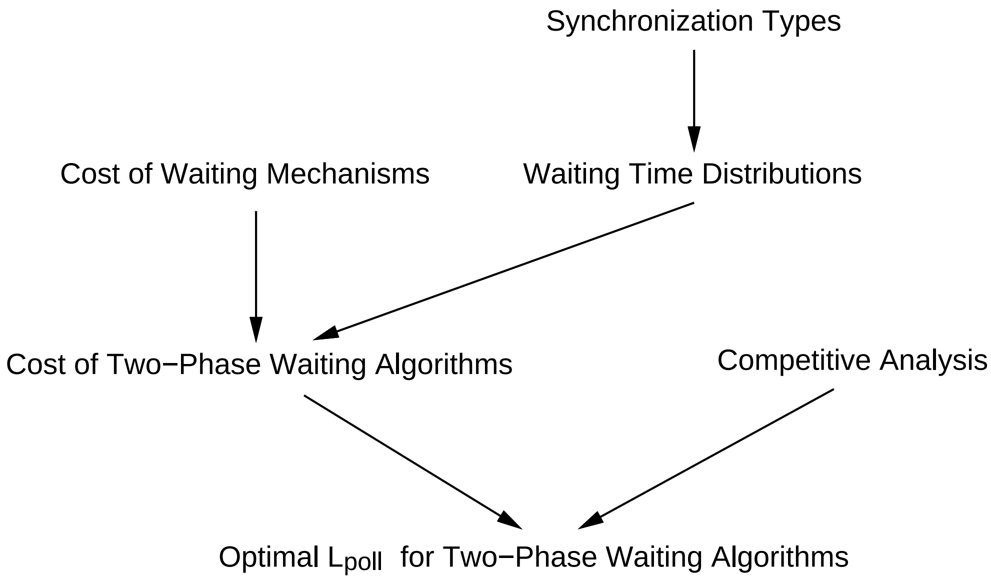


Figure 4.3: *Overview of method for analyzing the expected costs of waiting algorithms.*

An on-line algorithm can be considered as playing a game with an *adversary* that tries to select waiting times so as to maximize the cost of satisfying the requests. Using terminology in [26], a *strong adversary* is one that chooses requests depending on the choices made by the algorithm in satisfying previous requests. A *weak adversary* is one that chooses requests without regard to the previous choices made by the algorithm. Below, we introduce another form of adversary, called a *restricted adversary*, that is further constrained in its choice of requests.

It is well known that with a static choice of $L_{poll} = B$, a two-phase waiting algorithm is 2-competitive against a strong adversary: the worst possible scenario is to block after polling, incurring a cost of $2B$, when the optimal off-line algorithm would have blocked immediately, incurring a cost of B.

If we weaken the adversary and consider expected costs, a *dynamic* two-phase waiting algorithm can achieve lower competitive factors. In [26], Karlin *et al.* present a dynamic, randomized two-phase waiting algorithm with an expected competitive factor of $e/(e-1) \approx 1.58$ and prove this factor to be optimal for on-line algorithms against a weak adversary. They also prove that an adaptive algorithm that dynamically maintains waiting-time statistics can approach a competitive factor of 1.58 against a weak adversary.

We can further weaken the adversary by fixing the waiting time distribution and allowing it to control only the parameters of the distribution. For example, we can constrain waiting times to be exponentially distributed and allow the adversary to control only the arrival rate of the distribution. We term such an adversary a *restricted adversary*.

A restricted adversary models the situation where the waiting time distribution is fixed, but the parameters of the distribution depend on run-time factors. This situation commonly arises in practice. For example, although waiting times for producer-consumer synchronization may be exponentially distributed, the arrival rate of the exponential distribution may depend on the application and on run-time conditions.

Under restricted adversaries, static two-phase waiting algorithms can attain or approach the optimal on-line competitive factor of $e/(e-1)$. Our analysis determines optimal static choices of L_{poll} for exponentially and uniformly distributed waiting times. It shows that with exponentially distributed waiting times, a static algorithm with $\alpha = \ln(e-1)$ performs as well as any dynamic algorithm against a restricted adversary.

4.4.2 Expected Waiting Costs

In order to determine optimal settings of L_{poll}, we model the expected cost of a two-phase waiting algorithm as a function of waiting-time distributions. In the following analysis, $f(t)$ is the probability density function (PDF) of waiting times. ($f(t)$ is nonzero only for $t \geq 0$). L_{poll} is expressed as a multiple α of the cost of signaling B. That is, $L_{poll} = \alpha B$. We denote the cost of algorithm a as C_a, and its expected cost as $E[C_a]$.

The following equation gives the expected waiting cost for static two-phase waiting algorithms, where a polling mechanism is used for the first phase and a signaling mechanism for the second. A polling mechanism incurs a cost of t/β, while a signaling mechanism incurs a fixed cost B.

$$E[C_{\mathsf{2phase}/\alpha}] = \int_0^{\alpha\beta B} \frac{t}{\beta} f(t)dt + \int_{\alpha\beta B}^{\infty} (1+\alpha)Bf(t)dt \tag{4.1}$$

The first integral is the contribution to the expected waiting cost due to the probability that waiting times are less than $\alpha\beta B$ cycles. In this case, the waiting cost is simply the cost of polling, t/β. The second integral corresponds to the probability that the waiting time is more than $\alpha\beta B$ cycles, such that the waiting cost is L_{poll} plus B. $E[C_{\mathsf{poll}}]$ is derived by setting α to ∞, and $E[C_{\mathsf{signal}}]$ is derived by setting α to 0.

The following equation gives the expected cost of an optimal off-line algorithm that chooses between polling and signaling, and is derived by observing that the optimal algorithm polls if $t \leq \beta B$, and signals otherwise.

$$E[C_{\mathsf{Opt}}] = \int_0^{\beta B} \frac{t}{\beta} f(t)dt + \int_{\beta B}^{\infty} Bf(t)dt \tag{4.2}$$

4.4.3 Waiting Time Distributions and Synchronization Types

We consider three types of synchronization: producer-consumer, barrier, and mutual exclusion. We argue here that if we assume that arrivals of synchronizing threads are generated by a Poisson process, these synchronization types naturally result in exponentially and uniformly distributed waiting times. The Poisson assumption is a useful approximation of the behavior of many complex systems, and helps to make analysis tractable.

Producer-Consumer Synchronization Producer-consumer synchronization is performed between *one* producer and *one or more* consumers of the data produced. Examples of this type of synchronization include *future*s [22] and I-structures [6]. This form of producer-consumer synchronization is different from another form where only one consumer is allowed to consume the data. This second form of producer-consumer synchronization can be modeled as mutual-exclusion synchronization.

If we assume Poisson arrivals of producer threads, it immediately follows that waiting times for producer-consumer synchronization are exponentially distributed.

Barrier Synchronization Barrier synchronization ensures that all threads participating in a barrier have reached a point in a program before proceeding. The uniform distribution is a reasonable model for barrier waiting times. Such waiting times would arise if inter-barrier thread execution lengths are uniformly distributed within some time interval. We also show in [39] that if barrier arrivals are generated by a Poisson process, then waiting times approach a uniform distribution.

Mutual-Exclusion Mutual-exclusion synchronization provides exclusive access to data structures and critical sections of code. Assuming that lock waiters are not queued, waiting times at mutexes can be modeled by either an exponential or uniform distribution, depending on the distribution of lock-holding times. If lock-holding times are exponential, it follows that lock waiting times are also exponential.

If lock-holding times are fixed and deterministic, we have to differentiate between *new* waiters and *repeat* waiters. New waiters are freshly arrived lock requesters, and repeat waiters are lock requesters that re-contended unsuccessfully for the lock. These waiters experience different waiting times.

New waiters arrive at any time during the fixed interval when the lock is busy. If the new arrivals are Poisson, the waiting time for new waiters are uniformly distributed between 0 and the fixed lock-holding time. In contrast, repeat waiters have to wait for the entire duration of the lock holding time. Thus, the waiting time for repeat waiters is simply the fixed lock holding time. If we keep a history of lock holding times, it should be straightforward to decide whether to block repeat waiters.

If lock waiters are queued, we can use an $M/M/1//M$ queuing model to model waiting time distributions. Unfortunately, the resulting PDF of waiting times from such a model is sufficiently complex that it does not lend itself to a closed-form analysis. However, we note that under conditions of low lock contention, the queuing model predicts close to exponentially distributed waiting times. See [39] for a more detailed discussion.

4.5 Deriving Optimal Values for L_{poll}

The following analysis focuses on the exponential and uniform distributions as models for waiting time distributions. Section 4.6 presents empirical measurements of waiting times encountered in parallel applications that exhibit such waiting time distributions.

Using the equations for expected waiting costs and the models for waiting time distributions, we compute the expected competitive factors of static two-phase waiting algorithms. This allows us to derive optimal static values for L_{poll} for different waiting time distributions. In the following analysis, we express L_{poll} as a multiple, α, of the cost of signaling, B.

4.5.1 Exponentially Distributed Waiting Times

The following PDF models exponentially distributed waiting times,

$$f(t) = \lambda e^{-\lambda t} \tag{4.3}$$

where λ is the arrival rate of the synchronizing threads.

From Equations 4.1–4.3, we derive the following expressions for the expected costs of the always-poll (poll), always-signal (signal), static two-phase (2phase/α), and optimal off-line (Opt) waiting algorithms.

$$E[C_{\mathsf{poll}}] = \int_0^\infty \frac{t}{\beta}\lambda e^{-\lambda t} dt = \frac{1}{\lambda\beta} \tag{4.4}$$

$$E[C_{\mathsf{signal}}] = B \tag{4.5}$$

$$
\begin{aligned}
E[C_{\mathsf{2phase}/\alpha}] &= \int_0^{\alpha\beta B} \frac{t}{\beta}\lambda e^{-\lambda t}dt + \int_{\alpha\beta B}^{\infty}(1+\alpha)B\lambda e^{-\lambda t}dt \\
&= \frac{1}{\lambda\beta}(1-e^{-\lambda\alpha\beta B}) + Be^{-\lambda\alpha\beta B} \qquad (4.6)
\end{aligned}
$$

$$
\begin{aligned}
E[C_{\mathsf{Opt}}] &= \int_0^{\beta B} \frac{t}{\beta}\lambda e^{-\lambda t}dt + \int_{\beta B}^{\infty} B\lambda e^{-\lambda t}dt \\
&= \frac{1}{\lambda\beta}(1-e^{-\lambda\beta B}) \qquad (4.7)
\end{aligned}
$$

Comparing the expected performance of poll, signal and 2phase/α yields an interesting result. We expect that when arrival rates are high, poll performs better than signal. Conversely, when arrival rates are low, signal performs better than poll. The equations show that regardless of the arrival rate and L_{poll} the expected performance of static two-phase algorithms always falls in between the performance of poll and signal. More formally,

Theorem 1 *Under exponentially distributed waiting times, the expected costs of the algorithms* poll, signal, *and* 2phase/α *are ordered as*

$$
\begin{aligned}
E[C_{\mathsf{signal}}] \leq E[C_{\mathsf{2phase}/\alpha}] \leq E[C_{\mathsf{poll}}] &\quad \text{if} \quad \lambda\beta B \leq 1 \\
E[C_{\mathsf{signal}}] \geq E[C_{\mathsf{2phase}/\alpha}] \geq E[C_{\mathsf{poll}}] &\quad \text{if} \quad \lambda\beta B \geq 1
\end{aligned}
$$

Proof: By inspection, $E[C_{\mathsf{signal}}] \geq E[C_{\mathsf{poll}}]$ when $\lambda\beta B \geq 1$ and $E[C_{\mathsf{signal}}] \leq E[C_{\mathsf{poll}}]$ when $\lambda\beta B \leq 1$. Comparing $E[C_{\mathsf{signal}}]$ with $E[C_{\mathsf{2phase}/\alpha}]$ yields

$$
\begin{aligned}
E[C_{\mathsf{2phase}/\alpha}] \leq E[C_{\mathsf{signal}}] &\Leftrightarrow \frac{1}{\lambda\beta}(1-e^{-\lambda\alpha\beta B}) + Be^{-\lambda\alpha\beta B} \leq B \\
&\Leftrightarrow \lambda\beta B \geq 1.
\end{aligned}
$$

Comparing $E[C_{\mathsf{poll}}]$ with $E[C_{\mathsf{2phase}/\alpha}]$ yields

$$
\begin{aligned}
E[C_{\mathsf{2phase}/\alpha}] \leq E[C_{\mathsf{poll}}] &\Leftrightarrow \frac{1}{\lambda\beta}(1-e^{-\lambda\alpha\beta B}) + Be^{-\lambda\alpha\beta B} \leq \frac{1}{\lambda\beta} \\
&\Leftrightarrow \lambda\beta B \leq 1
\end{aligned}
$$

□

Empirical measurements (see Section 4.6) further indicate that two-phase algorithms are remarkably robust, and their performance is usually close to the better of poll and signal.

Next we observe that when $\lambda\beta B = 1$, the costs of all three algorithms are equal to B. That is, at the breakeven point where the arrival rate $\lambda = 1/\beta B$, the choice of L_{poll} has no effect on the expected cost of the two-phase algorithm. More formally,

Theorem 2 *Under exponentially distributed waiting times with $\lambda\beta B = 1$, the competitive factor of $\mathsf{2phase}/\alpha$ is $e/(e-1)$, regardless of the value of α.*

Proof: When $\lambda\beta B = 1$, we know from Theorem 1 that

$$E[C_{\mathsf{signal}}] = E[C_{\mathsf{2phase}/\alpha}] = E[C_{\mathsf{poll}}] = B$$

Therefore

$$\frac{E[C_{\mathsf{2phase}/\alpha}]}{E[C_{\mathsf{Opt}}]} = \frac{\lambda\beta B}{(1 - e^{-\lambda\beta B})} = \frac{e}{(e-1)}$$

□

This leads to the following corollary:

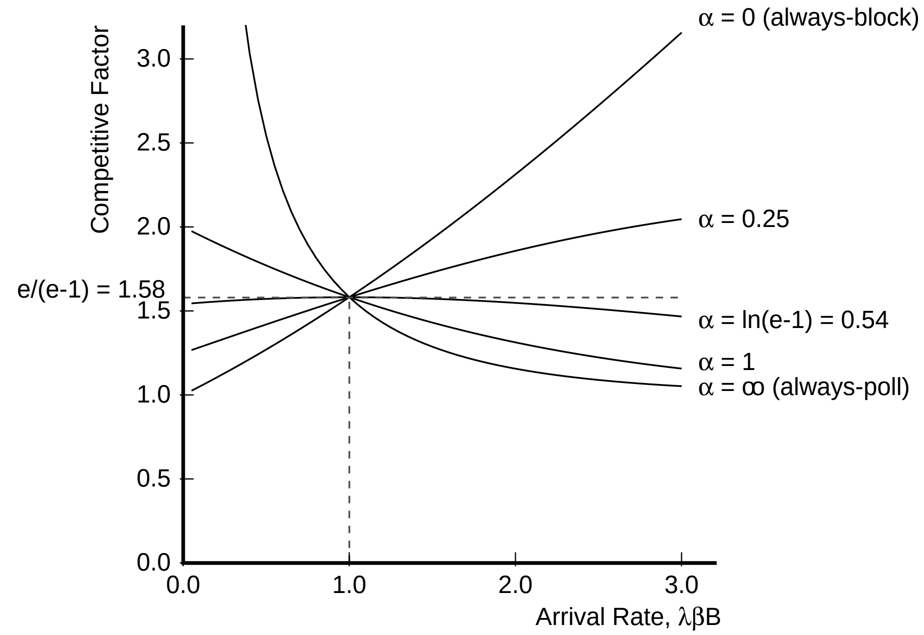
Corollary 1 *There exists a lower bound of $e/(e-1)$ on the competitive factor of any two-phase algorithm against strong, weak and restricted adversaries.*

Proof: The adversary picks exponentially distributed waiting times with $\lambda = 1/(\beta B)$. Regardless of the choice of α and regardless of whether the choice is made statically or dynamically, Theorem 2 implies the waiting cost is $e/(e-1)$ times that of an optimal off-line algorithm. It follows that one cannot construct a two-phase algorithm with a competitive factor lower than $e/(e-1)$. This competitive factor matches the lower bound obtained in [26] against a weak adversary. □

In light of this lower bound, the natural question to ask is whether a single static value for α can attain this lower bound under exponentially distributed waiting times. Surprisingly, the answer is yes, and the following theorem prescribes a value of α that yields optimal performance for exponentially distributed waiting times.

Theorem 3 *Under exponentially distributed waiting times with $\alpha = \ln(e-1)$, the competitive factor of two-phase waiting, $E[C_{\mathsf{2phase}/\alpha}]/E[C_{\mathsf{Opt}}]$, is at most $e/(e-1)$, regardless of the arrival rate, λ, of the distribution.*

Proof: Set $\alpha = \ln(e-1)$ in the equation for $E[C_{\mathsf{2phase}/\alpha}]/E[C_{\mathsf{Opt}}]$. This yields an equation for the competitive factor for two-phase waiting as a function of λ. Differentiate this equation with respect to λ to find the maximum. The resulting maximum competitive factor is $e/(e-1)$ at an arrival rate of $\lambda = 1/\beta B$. □



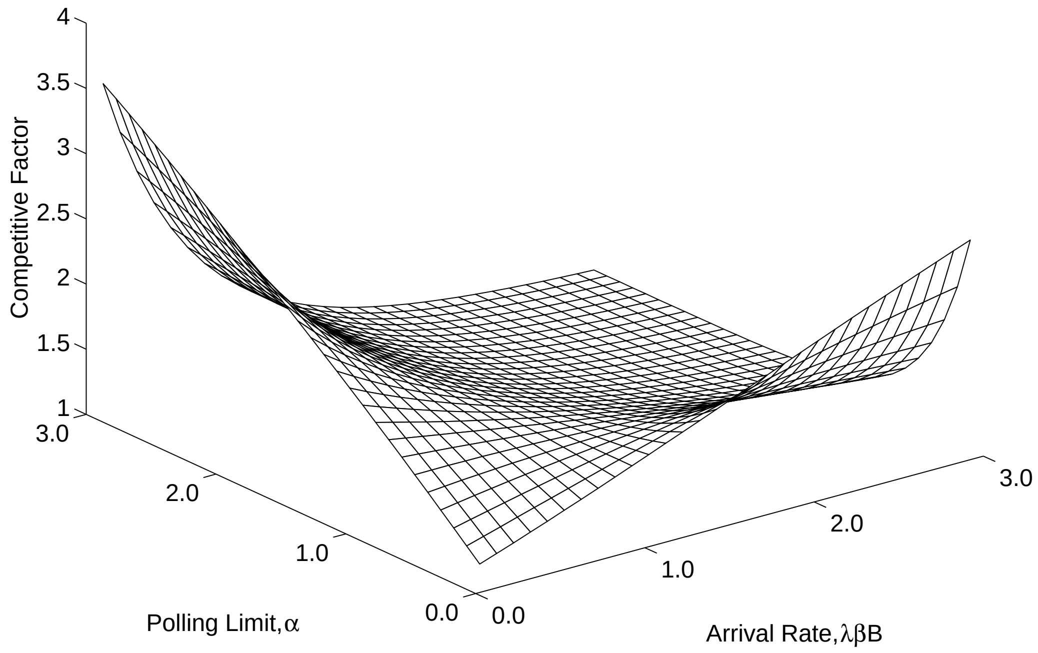


Figure 4.4: *Expected competitive factors under exponentially distributed waiting times, varying* λ *and* α*.*

These theorems are best illustrated by Figure 4.4. The 2-D and 3-D graphs plot the competitive factor of static two-phase waiting over a range of α and λ. The horizontal axis in the 2-D graph is in the direction of increasing λ and therefore shorter waiting times.

We see from the 2-D plot that the curves for finite non-zero values of α lie in between those of always switch-spin ($\alpha = \infty$) and always-block ($\alpha = 0$), as indicated by Theorem 1. We also see that all the curves intersect at a competitive factor of $e/(e-1)$ when $\lambda = 1/\beta B$ as indicated by Theorem 2. Lastly, we can see that the competitive factor is at most $e/(e-1)$ when $\alpha = \ln(e-1)$, as indicated by Theorem 3. Since actual values of λ are not relied on, this upper bound holds in the face of run-time uncertainty and feedback effects of the waiting algorithm on the waiting time as long as the waiting-time distributions is exponential.

These theorems imply that when waiting times are exponential, we should choose our waiting algorithm depending on knowledge of λ. If we know that $\lambda < 1/\beta B$, we should choose signal, otherwise we should choose poll. However, if we cannot reliably predict λ, we should choose 2phase/0.54 to obtain the best competitive factor of 1.58.

4.5.2 Uniformly Distributed Waiting Times

Here, we assume that waiting times are uniformly distributed between 0 and U. Repeating the previous analysis for exponentially distributed waiting times, we prove the following theorem for uniformly distributed waiting times.

Theorem 4 *Under uniformly distributed waiting times from $t = 0$ to U, with $\alpha = (\sqrt{5}-1)/2 \approx 0.62$, the competitive factor of two-phase waiting, $E[C_{\mathsf{2phase}/\alpha}]/E[C_{\mathsf{Opt}}]$, is at most $(\sqrt{5}+1)/2 \approx 1.62$, regardless of the parameter, U, of the distribution. Furthermore, if $\alpha \neq (\sqrt{5}-1)/2$, then the competitive factor under uniformly distributed waiting times is larger than $(\sqrt{5}+1)/2$.*

In other words, under uniformly distributed waiting times, a static two-phase algorithm with $\alpha = (\sqrt{5}-1)/2$ has a competitive factor no larger than $(\sqrt{5}+1)/2$, and no other value of α yields a lower competitive factor over the entire range of the parameter, U, of the uniform distribution. This result is illustrated in Figure 4.5. The horizontal axis of the 2-D graph is in the direction of decreasing U and therefore shorter waiting times.

Proof: Let waiting time be uniformly distributed from 0 to U. From Equations 4.1–4.2, we can derive the following expressions for the expected costs of static two-phase waiting algorithms and the optimal off-line algorithm.

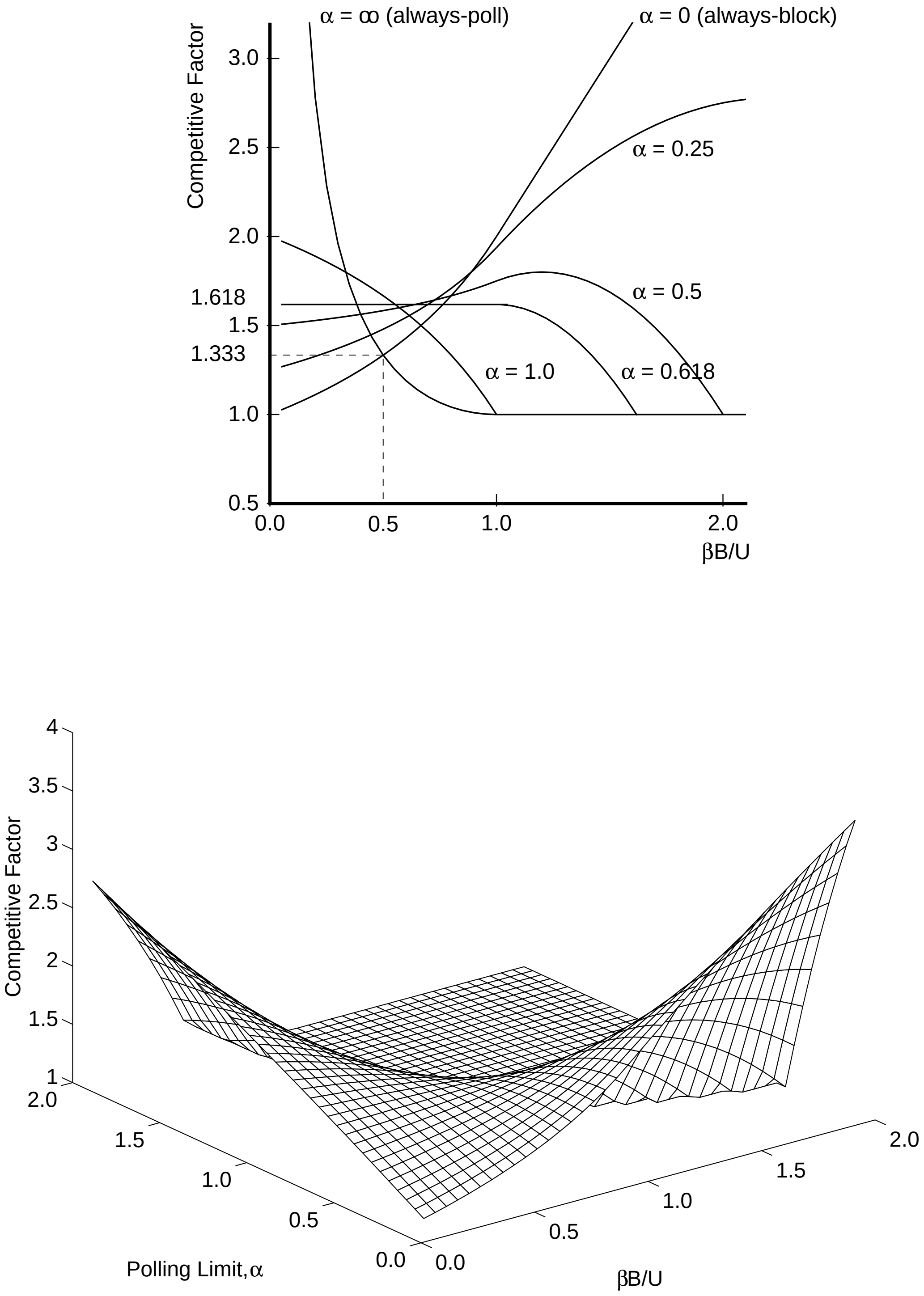


Figure 4.5: *Expected competitive factors under waiting times uniformly distributed between 0 and U, varying U and α.*

$$E[C_{\mathsf{2phase}/\alpha}] = \begin{cases} \int_0^U \frac{t}{\beta}\frac{1}{U}dt = \frac{U}{2\beta} & \text{if } U \leq \alpha\beta B \\ \\ \int_0^{\alpha\beta B} \frac{t}{\beta}\frac{1}{U}dt + \int_{\alpha\beta B}^{U}(1+\alpha)\frac{B}{U}dt = \\ \frac{1}{U}\left[(1+\alpha)BU - (1+\frac{\alpha}{2})\alpha\beta B^2\right] & \text{otherwise} \end{cases}$$

$$E[C_{\mathsf{Opt}}] = \begin{cases} \int_0^U \frac{t}{\beta}\frac{1}{U}dt = \frac{U}{2\beta} & \text{if } U \leq \beta B \\ \\ \int_0^{\beta B} \frac{t}{\beta}\frac{1}{U}dt + \int_{\beta B}^{U} \frac{B}{U}dt = \frac{1}{U}\left[BU - \frac{1}{2}\beta B^2\right] & \text{otherwise} \end{cases}$$

Let us consider the case when $\alpha \leq 1$. Substituting $x = U/\beta B$, we get the following expressions for the expected competitive factor, $c = E[C_{\mathsf{2phase}/\alpha}]/E[C_{\mathsf{Opt}}]$.

$$c = \begin{cases} 1 & \text{if } x \leq \alpha \\ \\ \left[2(1+\alpha)x - \alpha(\alpha+2)\right]/x^2 & \text{if } \alpha \leq x \leq 1 \\ \\ \left[2(1+\alpha)x - \alpha(\alpha+2)\right]/(2x-1) & \text{if } x \geq 1 \end{cases}$$

Also,

$$\frac{\partial c}{\partial x} = \begin{cases} 0 & \text{if } x \leq \alpha \\ \\ 2\left[\alpha(2+\alpha) - (1+\alpha)x\right]/x^3 & \text{if } \alpha \leq x \leq 1 \\ \\ \left[2(\alpha^2+\alpha-1)\right]/(2x-1)^2 & \text{if } x \geq 1 \end{cases}$$

In the range $x \geq 1$, $\frac{\partial c}{\partial x} = 0$ when either $x = \infty$ or $(\alpha^2 + \alpha - 1) = 0$. This implies that when $\alpha = (\sqrt{5}-1)/2$, the value of c is $(\sqrt{5}+1)/2$ over the entire range $x \geq 1$.

In the range $\alpha \leq x \leq 1$, $\frac{\partial c}{\partial x} = 0$ when either $x = \infty$ or $x = \alpha(2+\alpha)/(1+\alpha)$. Also, $\frac{\partial^2 c}{\partial x^2}$ is negative. These imply that when $\alpha = (\sqrt{5}-1)/2$, c has a maximum value of $(\sqrt{5}+1)/2$ at $x = 1$. Therefore, $c \leq (\sqrt{5}+1)/2$ when $\alpha = (\sqrt{5}-1)/2$.

We now have to show that no other setting of α yields a competitive factor of less than 1.62 over the entire range of U, so that α is the optimal setting for uniformly distributed waiting times.

As $x \to \infty$, c approaches $1 + \alpha$. Therefore the competitive factor is larger than $(\sqrt{5}+1)/2$ when $\alpha > (\sqrt{5}-1)/2$.

Now consider the case when $\alpha < (\sqrt{5}-1)/2$. In the range $x \geq 1$, $\frac{\partial c}{\partial x} < 0$ so that c monotonically decreases with x. Therefore the maximum value of c in this range is $(2-\alpha^2)$ when $x = 1$. Since $\alpha < (\sqrt{5}-1)/2 \Leftrightarrow (2-\alpha^2) > (\sqrt{5}+1)/2$, the theorem also holds for all $\alpha < (\sqrt{5}-1)/2$. □

The theorem says that we should choose our waiting algorithm using our knowledge of U. If we know that $U > 2\beta B$, we should choose signal, otherwise we should choose poll. Therefore, with accurate information about U we can attain a competitive factor of $4/3$ as illustrated in Figure 4.5.

However, it is hard to predict U since barrier waiting times are highly dependent on run-time factors [58]. If we cannot reliably predict U, we should choose 2phase/0.62 to obtain the best competitive factor of 1.62 (the golden ratio), as prescribed by Theorem 4, and as illustrated in Figure 4.5. This is close to the optimal on-line competitive factor of 1.58 against weak adversaries.

4.5.3 Summary

Let us summarize the results of the preceding analysis. The analysis shows that for exponentially distributed waiting times,

1. The performance of two-phase waiting always lies in between those of always-block and always-spin.

2. When a restricted adversary get to choose λ (the arrival rate), the competitive factor of static two-phase waiting has a lower bound of $e/(e-1)$. Furthermore, no dynamic algorithm can attain a lower competitive factor. Recall that this competitive factor is also optimal for on-line algorithms against weak adversaries.

3. A static value of $\ln(e-1)B$ for L_{poll} results in an algorithm that attains this lower bound of $e/(e-1)$ against a restricted adversary.

4. We should choose L_{poll} based on λ. If we know that $\lambda < 1/\beta B$, we should choose signal, otherwise we should choose poll. However, if we cannot predict λ, we should choose 2phase/0.54 to obtain the best competitive factor of $e/(e-1) \approx 1.58$.

For uniformly distributed waiting times, the analysis shows that

1. When a restricted adversary gets to choose U, the parameter of the uniform distribution, the competitive factor of static two-phase waiting has a lower bound of $(\sqrt{5}+1)/2$.

2. A static value of $\frac{1}{2}(\sqrt{5}-1)B$ for L_{poll} results in an algorithm that attains this lower bound of $(\sqrt{5}+1)/2$ against a restricted adversary. Furthermore, no other static choice of L_{poll} attains this competitive factor.

3. We should choose L_{poll} based on U. If we know that $U > 2\beta B$, we should choose signal, otherwise we should choose poll. If we cannot predict U, we should choose 2phase/0.62 to obtain the best competitive factor of $(\sqrt{5}+1)/2 \approx 1.62$.

4.6 Experiments

To show that static two-phase waiting algorithms work well in practice, and to corroborate the analysis of the previous section, we profiled the executions of several benchmark programs using various synchronization types on the Alewife simulator. Before describing the results of the experiments, we first describe the data that were collected, the benchmarks that were run, and the synchronization constructs that they used.

We collected several statistics from the simulations. First, we compiled *waiting-time profiles* that record the synchronization waiting times encountered in a program. These waiting-time profiles corroborate the waiting-time models that were developed in the previous section. They show that the exponential and uniform distributions are reasonable models for waiting times.

Second, we measured the total number of cycles consumed by the blocking routines. For an always-block waiting algorithm these cycles correspond to the waiting cost incurred while running a program. This statistic is useful in estimating the potential effect of a waiting algorithm on the running time of a benchmark, and allows us to speculate on the performance of waiting algorithms on larger machines where waiting overheads are expected to be more significant.

Third, we keep a count of the number of threads blocked during the execution of the program. We expect a two-phase algorithm to reduce the number of blocked threads, giving us some insight on how well the two-phase algorithm is performing relative to an always-block algorithm. Fourth, we measured the program execution time under each waiting algorithm.

4.6.1 Synchronization Constructs

The benchmarks use the following synchronization constructs that are representative of producer-consumer, barrier, and mutual-exclusion synchronization.

J-structures (Reusable I-structures)

A J-structure is a data structure for producer-consumer-style synchronization on vector elements which enables efficient fine-grained, data-level synchronization. It is implemented as a vector with full/empty bits associated with each vector slot. See [28] for further details.

A reader of a J-structure slot waits until the slot is full before returning the value. A writer of a J-structure slot writes a value to the slot, sets it to full, and releases all waiters for the slot. An empty vector slot doubles as the queue pointer for waiting readers. A write to a full slot signals an error. We allow a J-structure slot to be *reset*. A reset empties the slot, permitting multiple assignments. Reusing J-structure slots in this way allows efficient cache performance. J-structures can be used to implement I-structure [6] semantics.

Futures

Futures [22] are a method for specifying control parallelism. The expression *(future X)* specifies that the expression X may be executed in parallel with the current thread. If a thread is forked to evaluate X, the return value of *(future X)* is a placeholder for the value that will be eventually determined when the forked thread terminates.

Futures are a form of producer-consumer synchronization. The forked thread is responsible for producing the result of evaluating X. Consumer threads that need the result of X need to wait for the producer thread. The placeholder is an object that initially holds the queue of waiting consumers and eventually holds the result of evaluating X.

L-structures (Lock-able structures)

Like J-structures, an L-structure is implemented as a vector with full/empty bits associated with each vector slot. L-structures support three operations: a locking read, an unlocking write, and a non-locking read. A locking read waits until a slot is full before emptying the slot and returning the value. An unlocking write writes a value to an empty slot, and sets it to full, releasing any waiters. It is an error to perform an unlocking write to a full slot. A non-locking read returns the value found in a slot if full; otherwise it returns an invalid value.

An L-structure therefore allows mutually exclusive access to each of its slots. The locking and unlocking L-structure reads and writes are sufficient to implement M-structures [8]. L-structures are different from M-structures in that they allow multiple non-locking readers.

Semaphores

Semaphores are used to implement mutual-exclusion. A semaphore is implemented as a one-element L-structure. `semaphore-P` and `semaphore-V` are easily implemented using L-structure reads and writes.

Name of Benchmark	Synchronization Type	Matched/ Unmatched
MGrid	producer-consumer	settable
Jacobi	producer-consumer	settable
Factor	producer-consumer	unmatched
Queens	producer-consumer	unmatched
CGrad	barrier	settable
Jacobi-Bar	barrier	settable
FibHeap	mutual exclusion	settable
Mutex	mutual exclusion	settable

Table 4.2: *Benchmarks used for testing waiting algorithms.*

Barriers
Barriers ensure that all participating threads have reached a point in a program before proceeding. To avoid excessive traffic to a single location, and to distribute the enqueuing and release operations, we use software combining trees [57] to implement barriers.

4.6.2 Benchmarks

The experiments use benchmarks that are representative of producer-consumer, barrier, and mutual-exclusion synchronization. Table 4.2 lists the benchmarks and indicates the synchronization types in each of the benchmarks.

Blocking only makes sense if there is another runnable thread to execute. Therefore we differentiate between the case where the number of threads is perfectly matched to the number of processors, and the case where there are more threads than processors. To ease discussion, let us say that a program is *matched* if the number of concurrently runnable threads assigned to any processor never exceeds the number of hardware contexts on that processor; otherwise the program is *unmatched*. Table 4.2 indicates whether a benchmark is matched or unmatched.

We describe the communication and synchronization characteristics of each of the benchmarks below.

MGrid applies the multigrid algorithm to solving Poisson's equation on a 2-D grid. Communication is nearest-neighbor except during shrink and expand phases. The 2-D grid is partitioned into subgrids, and a thread is assigned to each subgrid. Borders of each subgrid are implemented as J-structures to allow fine grain synchronization with neighbors. The J-structures are reset between iterations.

Jacobi performs Jacobi relaxation for solving Poisson's equation on a 2-D grid. Each thread is responsible for one grid point, and neighboring grid points are mapped onto neighboring processors. The grid is allocated uniformly so that only nearest-neighbor communication is necessary. J-structures are used to synchronize neighboring threads. The grain size of each thread is purposely made very small in order to expose the effects of synchronization as they become significant.

Factor computes the largest prime factors of each integer in a given range of integers, and accumulates them. The synchronization structure of the program can be most easily viewed as a recursive function call tree with synchronization occurring at each node of the tree. The program was dynamically partitioned with lazy task creation [44].

Queens solves the n-queens problem: given an $n \times n$ chess board, place n queens such that no two queens are on the same row, column, or diagonal. A search of all possible solutions is made and this particular benchmark was run with $n = 9$ and with lazy task creation. Queens has similar synchronization characteristics to Factor.

CGrad is the conjugate gradient numerical algorithm for solving systems of linear equations. In this benchmark, the algorithm is used to solve Poisson's equation on a 2-D grid. Each iteration of CGrad involves global accumulates and broadcasts which are implemented using a software combining tree. These accumulates and broadcasts also serve as barriers between phases.

Jacobi-Bar solves exactly the same problem as Jacobi, but uses a global barrier between iterations for synchronization instead of J-structures. Like in Jacobi, only nearest neighbor communication is necessary within an iteration.

CountNet tests an implementation of a counting network [7]. Threads repeatedly try to increment the value of a counter through a bitonic counting network so as to reduce contention and allow parallelism. Threads acquire and release mutexes at each network node as they traverse the network.

`FibHeap` tests an implementation of a scalable priority queue based on a Fibonacci heap [24]. Mutexes are used to ensure atomic updates to the heap. Scalability is achieved by distributing mutexes throughout the data structure. This avoids points of high lock contention and allows parallelism. The test involves repeatedly executing `insert` and `extract-min` operations on the priority queue.

`Mutex` is a synthetic benchmark that monitors the performance of mutexes under varying loads. Worker threads are distributed evenly throughout the machine and each thread runs a loop that with some fixed probability acquires a mutex, executes a critical section, then releases the mutex.

4.7 Experimental Results

This section presents the results of executing the benchmarks on a simulation of a 64-processor Alewife machine. Switch-spinning was used as the polling mechanism, while blocking was used as the signaling mechanism. We first present the waiting-time profiles and compare them with the proposed models for the three synchronization types. We then present the resulting program execution times under different waiting algorithms.

4.7.1 Waiting-Time Profiles

The waiting-time profiles are gathered by monitoring the waiting times for each failed synchronization. A number of the profiles approximate an exponential distribution. Whenever this is so, a semi-log plot is used so that the exponential distribution is easily recognizable as a linear set of points. Linear regressions on the log values of the waiting time frequencies are also plotted. This corresponds to fitting exponential curves through the original set of points. Outliers with frequencies less than 10 were pruned in the regressions.

Producer-Consumer Synchronization Figures 4.6 and 4.7 present semi-log plots of waiting-time profiles obtained from benchmarks with producer-consumer synchronization. These profiles support the use of exponential waiting times in our competitive analysis of waiting algorithms.

We see from the plots that the waiting times are indeed largely exponentially distributed. However, there is some deviation for short waiting times in the unmatched versions of MGrid and Jacobi. We believe this is due to the effect of blocking on waiting times. Blocked

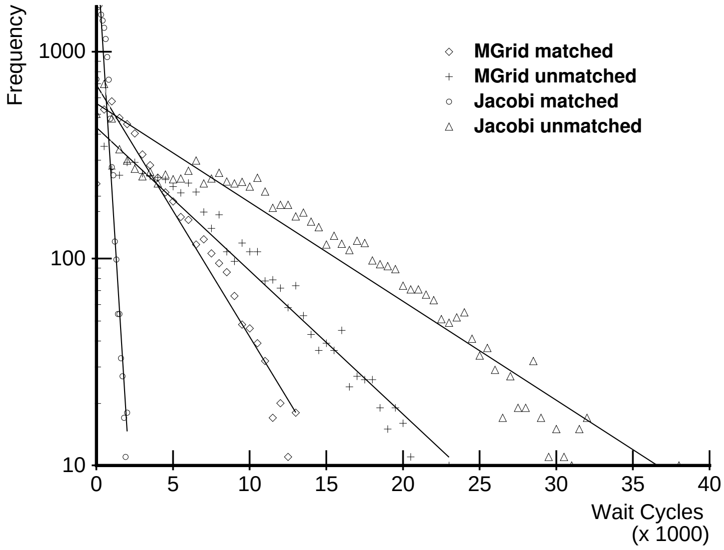


Figure 4.6: *Measured waiting times for J-structure readers.*

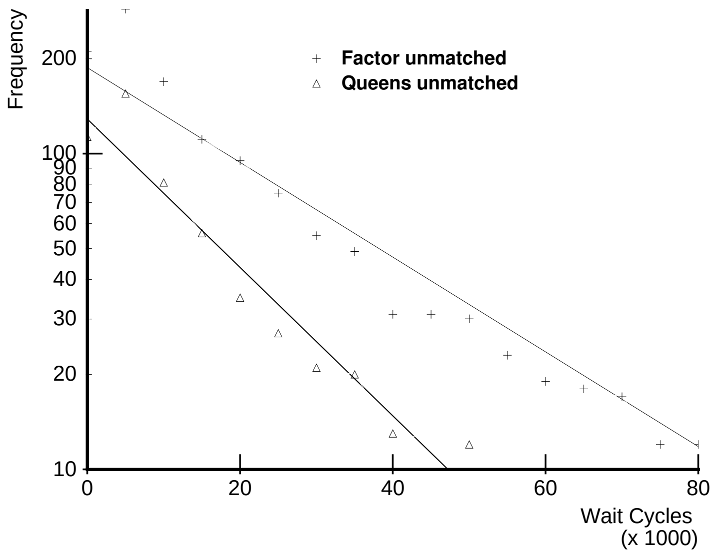


Figure 4.7: *Measured waiting times for futures.*

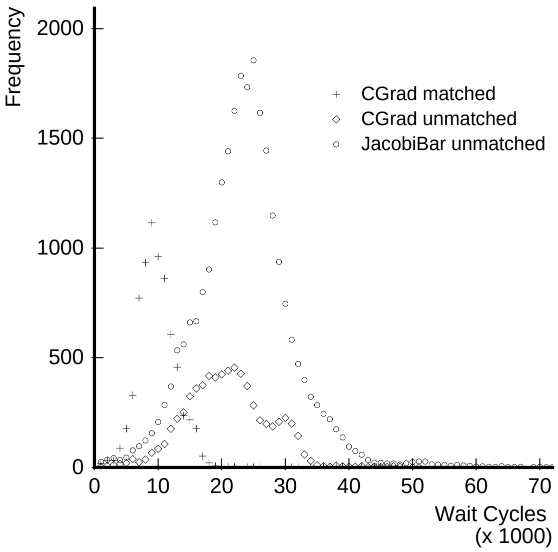


Figure 4.8: *Measured barrier wait times for* CGrad *and* Jacobi-Bar.

threads experience some delay before resuming execution. Since a blocked thread might be a producer that is waited on by some consumer threads, this delay can cause a fraction of waiting times to be skewed upward.

Although it would be premature to conclude from these results that producer-consumer waiting times are exponentially distributed, the data show the existence of parallel programs that exhibit such waiting times. In such cases, a static setting of $L_{poll} = 0.54$ should yield better performance.

Barrier Synchronization Figure 4.8 presents the waiting-time profiles for CGrad and Jacobi-Bar. Although our model suggests that barrier waiting times should be uniformly distributed, the waiting-time profiles do not support this hypothesis. This deviation is due to the overhead of the software combining tree barrier in this experiment. An arrival at a combining tree barrier has to traverse some part of the combining tree before waiting. The traversal is not considered as waiting time.

To filter out this software overhead, we ran a version of Jacobi-Bar with a simple counter implementation of barriers, thereby eliminating the combining tree. We executed this benchmark on a simulation of an idealized, one-cycle access memory system to eliminate the effect of hardware contention on this simple barrier implementation. Figure 4.9 presents the resulting waiting-time profile which is close to uniform except at the tails.

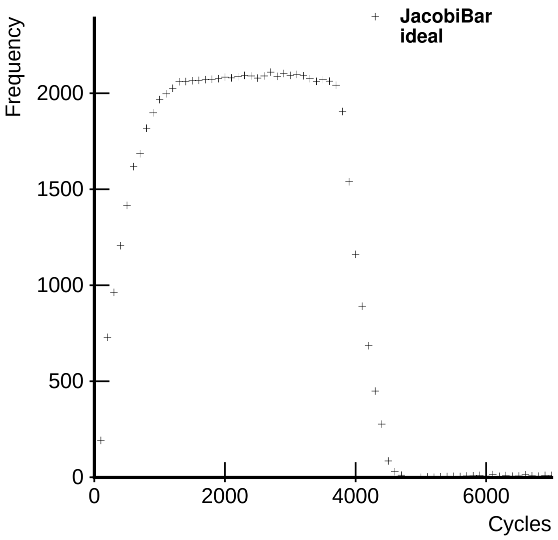


Figure 4.9: *Measured barrier waiting times for* Jacobi-Bar *on an ideal memory system.*

Mutual Exclusion Synchronization Figures 4.10 and 4.11 present the measured waiting times for the mutual-exclusion benchmarks. The waiting time is measured as the time from when a thread first fails to acquire the mutex to when that same thread successfully acquires the mutex[1].

The waiting times for FibHeap and Mutex appear to be exponential. However, although the waiting times for CountNet have exponential tails, the shorter waiting times in that benchmark deviate from an exponential distribution.

4.7.2 Application Performance

This section presents the program execution statistics of the benchmarks. These benchmarks were run on a simulation of a 64-processor Alewife machine with multiple hardware contexts. We compare the execution times of the benchmarks under the following waiting algorithms: two-phase waiting with $L_{poll} = B$ (2phase/1), always-poll (poll), and always-signal (signal). (For poll, L_{poll} is actually limited to 50000 cycles to implement a timeout mechanism for deadlock avoidance.)

Figures 4.12–4.14 present the execution times of the benchmarks for each of the synchronization types, normalized to the always-block waiting algorithm. We see from the

[1] Another possible measure of lock waiting time would be the time from when a thread first fails to acquire the mutex to when the mutex is released by the lock holder.

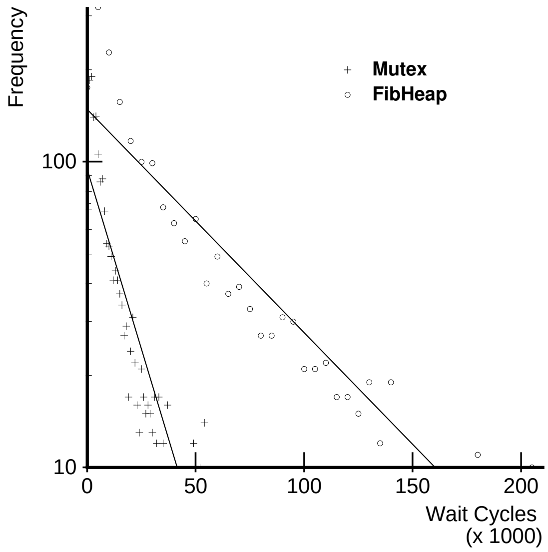


Figure 4.10: *Semi-log plot of measured mutex waiting times in* FibHeap *and* Mutex*.*

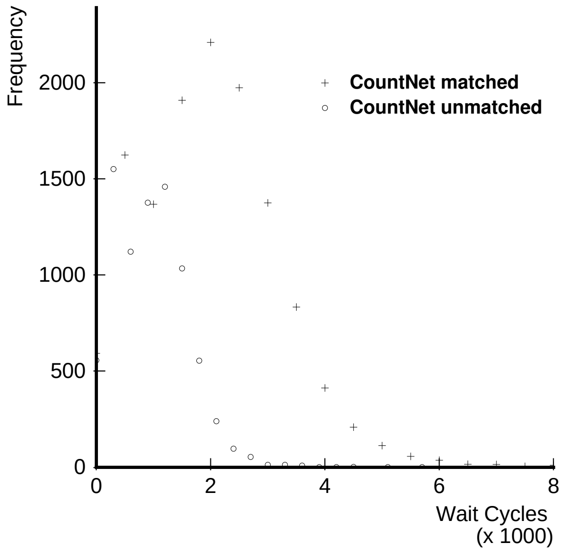


Figure 4.11: *Measured mutex waiting times in* CountNet*.*

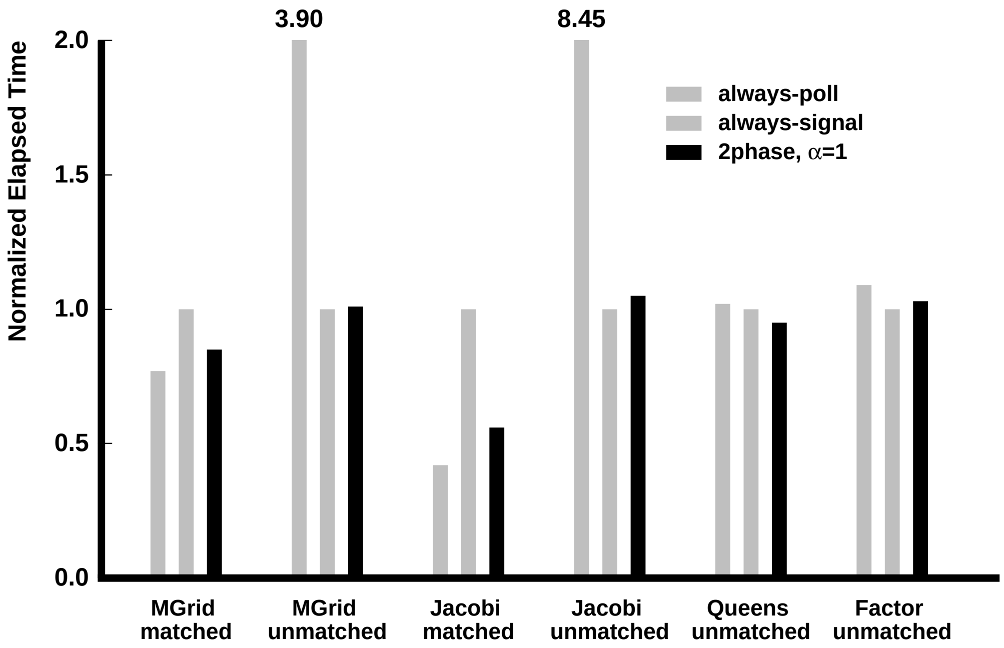


Figure 4.12: *Execution times for producer-consumer synchronization benchmarks under different waiting algorithms.*

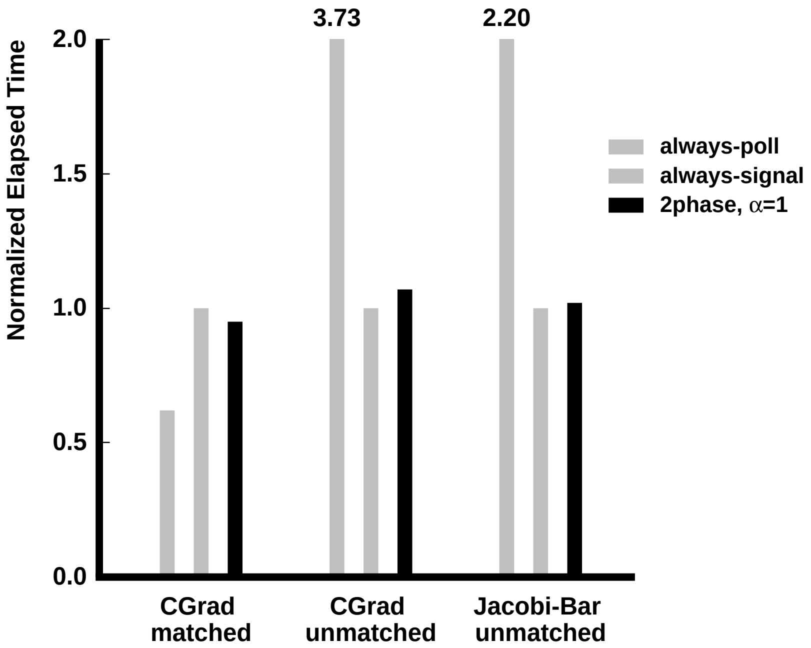


Figure 4.13: *Execution times for barrier synchronization benchmarks under different waiting algorithms.*

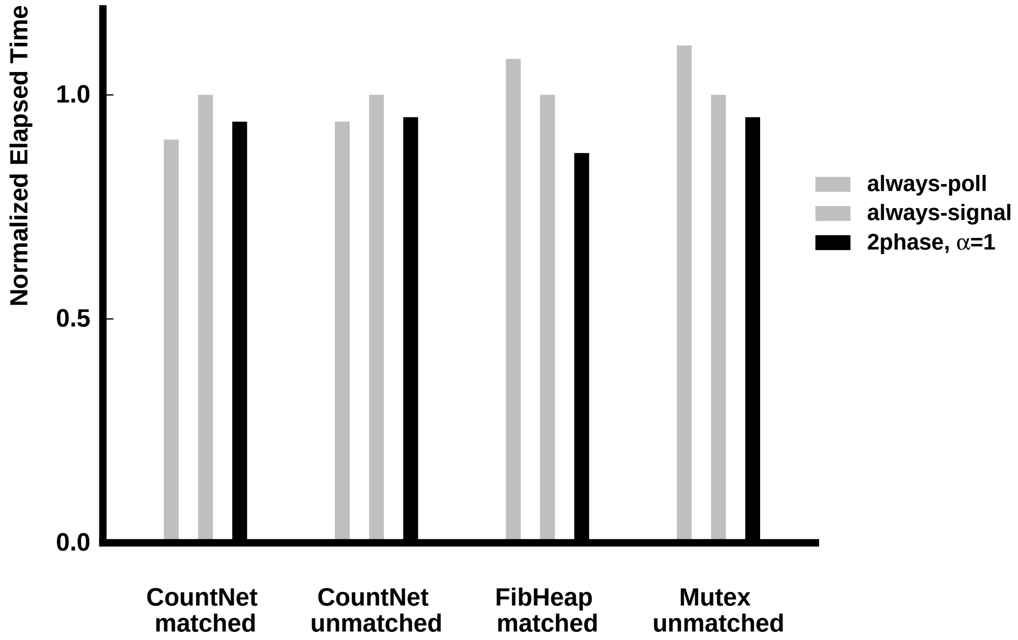


Figure 4.14: *Execution times for mutual-exclusion synchronization benchmarks under different waiting algorithms.*

graphs that the choice of waiting mechanisms makes a substantial difference in the execution times of the benchmarks.

Always-poll results in pathological performance in the unmatched producer-consumer and barrier synchronization benchmarks because polling threads can monopolize the processor and prevent a unloaded producer thread or barrier arrival from running. Mutual exclusion synchronization does not face this problem because Alewife's run-time system never unloads a lock holder. Even ignoring these pathological cases, the choice of waiting mechanisms can result in a performance difference of nearly 2.4 times between the best and worst cases (*cf.* Jacobi unmatched).

Despite the wide variance of run-time conditions across all the benchmarks, the two-phase waiting algorithm is always within 53% of the best waiting algorithm. If we disregard the matched program runs, where blocking is not beneficial, the two-phase waiting algorithm is within 6.6% of the best algorithm. This shows that two-phase waiting is extremely robust and performs close to the best static choice of waiting mechanisms across all the benchmarks. Most importantly, it never results in pathologically bad performance.

Let us consider the results for each of the synchronization types in more detail.

Producer-Consumer Synchronization Table 4.3 summarizes the detailed simulation results for barrier synchronization. Since waiting-time profiles for producer-consumer synchronization approximate an exponential distribution, we expect the performance of 2phase/1 to lie in between signal and poll (see Theorem 1). This is indeed the case[2], but more importantly, the measured performance of 2phase/1 is not far from the best algorithm in each case. 2phase/1 has the best overall performance among the three waiting algorithms.

poll encounters deadlock and times out in unmatched MGrid and Jacobi and thus performs poorly. This problem with deadlock is not present for unmatched Queens and Factor because they are dynamically partitioned with lazy task creation [44]. signal performs reasonably well except for matched Jacobi which has very short waiting times.

Barrier Synchronization Table 4.4 summarizes the detailed simulation results for barrier synchronization. Because of the nature of barrier synchronization, waiting times at barriers are likely to be long: a waiting thread is likely to be held up for a large number of other threads. In the benchmarks, the waiting-time profiles presented above indicate that most of the waiting times were longer than the blocking overhead. We see the effect of this in the performance figures in Table 4.4, where signal performs best in the unmatched programs. The number of blocked tasks also confirm that most of the waiting times are longer than B. This suggests that we should use signal at barriers unless we know that the program is matched. poll runs into deadlock for the unmatched programs.

Nevertheless, 2phase/1 performs quite well and is within 6.6% of the performance of signal. We can do better if we have some indication of the number of arrivals at the barrier. We cannot rely on the availability of a global count of arrivals in large-scale machines because that would limit the scalability of the barrier algorithm. However, for tournament-style tree barriers, we know that waits near the root of the tree should be shorter than waits near the leaves. Accordingly, we can use an always-signal algorithm for the lower sections of the tree and a two-phase algorithm for the upper sections.

Mutual Exclusion Table 4.5 summarizes the detailed simulation results for mutual-exclusion synchronization. In the mutual-exclusion benchmarks, deadlock is not an issue, even in unmatched conditions, because lock holders are never descheduled. 2phase/1 performs well in both matched and unmatched CountNet and performs best in FibHeap and

[2] 2phase/1 performs best in Queens because of an interaction with the scheduler and lazy task creation which resulted in a better partitioning of the program.

Benchmark	Con-texts	Waiting Algorithm	Runtime (Kcycles)	Normalized Runtime	Wait Ovh.[1]	Blocked Threads
MGrid matched	4	signal	2,251	1.0	5%	6,365
	4	2phase/1	1,918	0.85		1,769
	4	poll	1,731	0.77		4
MGrid unmatched	2	signal	1,865	1.0	7%	6,953
	2	2phase/1	1,885	1.01		5,150
	2	poll	7,273	3.90		4,613
Jacobi matched	4	signal	930	1.0	21%	12,818
	4	2phase/1	524	0.56		1,144
	4	poll	390	0.42		440
Jacobi unmatched	4	signal	719	1.0	21%	9,931
	4	2phase/1	757	1.05		7,756
	4	poll	6,075	8.45		4,399
Queens unmatched	4	signal	458	1.0	3%	655
	4	2phase/1	434	0.95		300
	4	poll	467	1.02		0
Factor unmatched	4	signal	769	1.0	5%	1,561
	4	2phase/1	790	1.03		913
	4	poll	841	1.09		19

[1]as percentage of runtime

Table 4.3: *Performance figures for producer-consumer synchronization.*

Benchmark	Con-texts	Waiting Algorithm	Runtime (Kcycles)	Normalized Runtime	Wait Ovh.[1]	Blocked Threads
CGrad matched	4	signal	1,052	1.0	11%	7,478
	4	2phase/1	999	0.95		7,161
	4	poll	654	0.62		2
CGrad unmatched	2	signal	1,048	1.0	11%	7,625
	2	2phase/1	1,118	1.07		7,309
	2	poll	3,905	3.73		3,714
Jacobi-Bar unmatched	4	signal	1,592	1.0	23%	26,880
	4	2phase/1	1,617	1.02		25,820
	4	poll	3,497	2.20		14,395

[1]as percentage of runtime

Table 4.4: *Performance figures for barrier synchronization.*

Benchmark	Con-texts	Waiting Algorithm	Runtime (Kcycles)	Normalized Runtime	Wait Ovh.[1]	Blocked Threads
CountNet matched	4	signal	1,378	1.0	9%	10,502
	4	2phase/1	1,293	0.94		1,913
	4	poll	1,242	0.90		276
CountNet unmatched	2	signal	1,298	1.0	8%	7,646
	2	2phase/1	1,241	0.95		1,202
	2	poll	1,224	0.94		43
FibHeap matched	4	signal	2,430	1.0	23%	7,882
	4	2phase/1	2,117	0.87		7,332
	4	poll	2,617	1.08		581
Mutex unmatched	4	signal	612	1.0	19%	4.429
	4	2phase/1	583	0.95		1,652
	4	poll	678	1.11		0

[1] as percentage of runtime

Table 4.5: *Performance figures for mutual-exclusion synchronization.*

Mutex. Again, this demonstrates the robustness of two-phase waiting. poll unexpectedly performs worst even in matched conditions in FibHeap. We will explain these observations here.

Lock contention was low in CountNet, and we know that a large number of waits were short from the waiting-time profiles above and by comparing the number of blocked threads for signal and 2phase/1. Under such conditions, poll performs best and signal worst, with 2phase/1 close to poll. However, since the waiting times are not exponential nor uniform, we cannot match these performance results with our theoretical analysis.

Lock contention was high in FibHeap and Mutex. The bad performance of poll in these benchmarks is due to the effect of contention. Because of the use of non-queuing locks in the benchmarks, a lock release immediately causes all polling waiters to re-contend for the lock, causing detrimental hot-spot contention. All the released waiters try to acquire the lock at once, exacerbating the waiting times at that lock. This is an example where an intelligent protocol selection algorithm would have improved the performance of poll by choosing a queuing protocol.

Because blocked waiters have to be rescheduled before they re-contend for the lock, signal actually helps avoid the detrimental effect of bursty lock requests. This allows signal to actually perform better than poll, even in matched FibHeap. 2phase/1 works best because it naturally polls on lightly contended locks and blocks on highly contended locks, combining the best of both worlds, an advantage not predicted by the theoretical models.

4.7.3 Changing L_{poll}

In the above results, L_{poll} was set to be equal to the cost of blocking. However, the preceding theoretical analysis indicates that setting L_{poll} to $0.54B$ will yield a more robust algorithm when waiting times are exponential, while setting L_{poll} to $0.62B$ will yield a more robust algorithm when waiting times are uniform.

While we would like to empirically confirm that the prescribed settings for L_{poll} lead to optimal competitive factors, doing so would require an infeasible amount of simulation. We would have to run a large set of benchmarks ranging over the possible values of the waiting-time distribution parameters. However, we attempt to lend some support to the theoretical results by taking some measurements with $L_{poll} = 0.5B$.

We experiment with two of the producer-consumer benchmarks (MGrid and Jacobi) under unmatched conditions. Table 4.6 reproduces the results presented earlier, and includes results for 2phase/0.5. We observe that a shorter polling phase results in better performance than 2phase/1 in MGrid and Jacobi because producer arrival rates were low. Under such conditions, *i.e.*, when $\lambda < 1/\beta B$, our theoretical analysis predicts that 2phase/0.5 will perform better than 2phase/1. Karlin *et al.* [25] also observe by analyzing measured waiting-time profiles that setting L_{poll} to $0.5B$ can result in lower waiting costs.

Surprisingly, 2phase/0.5 also performed better than signal. We think that this effect is due to the possibility of not finding a runnable thread to execute after blocking a thread, which violates the assumption made in the theoretical analysis. This would cause signal to unnecessarily block more threads compared to two-phase waiting. Also, the waiting algorithm itself may affect the waiting times.

4.8 Summary

This chapter explores the possibility of reducing the cost of waiting for synchronization by using a signaling waiting mechanism to overlap the waiting time with other computation. However, since signaling incurs a significant fixed cost, a run-time algorithm is needed to choose between polling and signaling mechanisms.

We transform the problem of choosing between waiting mechanisms into one of choosing the right value of L_{poll} for a two-phase waiting algorithm. In the interest of minimizing

Benchmark	Con-texts	Waiting Algorithm	Runtime (Kcycles)	Normalized Runtime	Wait Ovh.[1]	Blocked Threads
MGrid unmatched	2	signal	1,865	1.0	7%	6,953
	2	2phase/0.5	1,817	0.97		5,488
	2	2phase/1	1,885	1.01		5,150
	2	poll	7,273	3.90		4,613
Jacobi unmatched	4	signal	719	1.0	21%	9,931
	4	2phase/0.5	699	0.97		8,437
	4	2phase/1	757	1.05		7,756
	4	poll	6,075	8.45		4,399

[1]as percentage of runtime

Table 4.6: *Performance figures for* $L_{poll} = 0.5B$.

the run-time overhead of two-phase waiting, we focus on methods to choose L_{poll} statically.

Exploiting the fact that waiting times tend to be randomly distributed according to some waiting-time distribution, this chapter prescribes static values of L_{poll} that result in close to optimal competitive factors. In particular, it proves that if waiting times are exponentially distributed, then a static choice of $L_{poll} = 0.54B$ yields a 1.58-competitive waiting algorithm. It also proves that if waiting times are uniformly distributed, then a static choice of $L_{poll} = 0.62B$ yields a 1.62-competitive waiting algorithm. In practice, these results indicate that a static two-phase waiting algorithm should poll for about half the cost of signaling rather than the entire cost of signaling.

We ran some application programs on a simulation of the Alewife machine, and measured synchronization waiting times and program execution times under various waiting algorithms. The waiting-time profiles of a number of the programs are exponentially distributed. The execution time statistics show that two-phase waiting results in performance that is close to the best static choice of waiting mechanisms. The experiments also show that always-block is an acceptable waiting algorithm, and that always-poll is a poor waiting algorithm when there are more threads than processors.

Chapter 5

Related Work

The field of multiprocessor synchronization has been extensively studied. This chapter overviews related research on reducing synchronization costs. It first describes research on designing efficient algorithms for synchronization operations. It then describes other complementary approaches to reducing the cost of synchronization. Such approaches include program restructuring and multithreading. Finally, it describes research that uses a concept similar to $\mathcal{C}$-serializability and consensus objects for enhancing the performance of concurrent dictionary search operations.

5.1 Synchronization Algorithms

Previous research on designing efficient algorithms to minimize the cost of synchronization operations focuses on three approaches:

1. Scalable synchronization algorithms that perform well under high contention.
2. Waiting algorithms to minimize the cost of waiting for synchronization.
3. Adaptive, run-time methods.

We describe research in each of these areas in turn.

5.1.1 Scalable Synchronization Algorithms

Spin locks and barriers are commonly used to synchronize shared-memory programs. With the advent of larger multiprocessors, it became apparent that simple, centralized algorithms for spin locks and barriers scale poorly. The problem is twofold. First, contention at the centralized memory locations causes memory latencies to increase drastically. Second, the centralized nature of the algorithms removes any opportunity for parallelism by sequentializing accesses. In response to this problem of scaling to high contention levels, a recent area of research focuses on designing scalable algorithms that perform well under high contention.

Research on spin locks by Anderson [5], Mellor-Crummey and Scott [43], and Graunke and Thakkar [19] show that the best approach to implementing spin locks under high contention is to enqueue lock waiters and service them one at a time. This prevents lock waiters from simultaneously recontending for the lock and reduces the detrimental effects of memory contention.

Mellor-Crummey and Scott measured the performance of a number of scalable spin-barrier algorithms in [43]. Their results prescribe using a combining tree or butterfly network to combine arrival information and to signal barrier completion. Combining reduces contention and allows the algorithm to proceed in parallel.

Observing that mutual exclusion has the undesired effect of serializing processes, Gottlieb *et al.* [18] suggest a method of avoiding serialization by using a fetch-and-op operation. The advantage of fetch-and-op is that concurrent fetch-and-op operations to a single variable can be combined and can proceed in parallel. Goodman *et al.* [15] present a combining tree algorithm to compute fetch-and-op in parallel.

The price of using these scalable algorithms is that they typically have a higher protocol cost than simpler algorithms under low contention. In effect, these algorithms trade off performance at low contention for performance under high contention. These scalable algorithms are optimized for high contention although, in practice, the level of contention can (and should) be much less than the maximum number of processors. Nevertheless, this research has been useful in providing synchronization protocols to be selected at run-time by a protocol selection algorithm.

The experimental data in the research on scalable synchronization algorithms are based on purely synthetic benchmarks with static levels of contention. This thesis provides additional data on the performance these scalable algorithms in benchmarks with dynamically changing contention levels and in application programs.

5.1.2 Waiting Algorithms

Another area of research focuses on reducing waiting cost by overlapping waiting time with other useful computation. Research in this area has designed waiting algorithms that make intelligent run-time choices between spinning and blocking.

Ousterhout first proposed the two-phase waiting algorithm in his Medusa operating system [47]. The operating system implements two-phase waiting with a user-settable L_{poll}. In a study of multiprocessor scheduling algorithms, Lo and Gligor [40] found that use of two-phase waiting improves the performance of group scheduling when L_{poll} is set in between B and $2B$, where B is the cost of blocking.

This thesis shows that the effectiveness of two-phase waiting depends on both the distribution of waiting times and the setting of L_{poll}. Zahorjan *et al.* [58] studied the effect of data dependence and multiprogramming on waiting times for locks and barrier synchronization, and showed that waiting times can be highly dependent on run-time factors. They conclude that data dependence and multiprogramming does not significantly alter lock waiting times. However, for barrier synchronization, both data dependence and multiprogramming lead to sharply increased waiting times.

Research by Karlin *et al.* [26] focuses on selecting L_{poll} to optimize the performance of two-phase waiting. They present a randomized two-phase waiting algorithm, where the length of the polling phase is randomly picked from a predetermined probability distribution. The randomized algorithm achieves an expected competitive factor of $e/(e-1) \approx 1.58$. In a separate paper [25], Karlin *et al.* performed an empirical study of several techniques for determining L_{poll} in two-phase waiting algorithms for mutual exclusion locks. In this thesis, we show how to statically select L_{poll} so as to achieve close to the optimal on-line competitive factor of $e/(e-1)$.

5.1.3 Adaptive Methods

Certainly, the idea of run-time adaptivity to optimize performance is not new. Here, we describe some recent research in using adaptivity to improve the performance of operating system functions and synchronization operations.

Reconfigurable Operating Systems Mukherjee and Schwan [45] provide an overview of reconfigurable operating systems. The general idea is to provide hooks into an operating system so that application programs can dynamically control certain parameters of operating system services and improve performance. Mukherjee and Schwan present a model for adaptive operating system objects that adapt to run-time conditions, either automatically, or through user control. As an example, they implement a class of multiprocessor locks that they term *adaptive locks* and show that the added run-time cost of dynamic configuration is outweighed by the ensuing performance gains.

While Mukherjee and Schwan's adaptive lock and the reactive lock algorithm in this thesis both attempt to improve the performance of locks through adaptivity, there is a significant difference in the two approaches. Their adaptive lock allows scheduling and waiting policies of the lock to be reconfigured, but they do not go so far as to change the protocol in use. Reactive synchronization algorithms take a more general approach and deal with the harder problem of allowing the synchronization protocol itself to be changed.

Adaptive Mutual Exclusion Recent research by Yang and Anderson [55], and Choy and Singh [13] designed adaptive algorithms for mutual exclusion in the context of shared-memory multiprocessors that provide only atomic read and write primitives. They tackle the classic mutual exclusion problem of reducing the time complexity of implementing mutual exclusion with only atomic reads and writes. With this constraint, the best known mutual exclusion algorithms are either fast in the absence of contention but scale poorly, or slow in the absence of contention but scale well.

Yang and Anderson designed an algorithm that adaptively selects between Lamport's fast mutual exclusion algorithm [35] and a scalable algorithm of their design. It selects Lamport's algorithm when there is absolutely no contention, and the scalable algorithm when any contention is detected.

Choy and Singh use a *filter* as a building block for constructing mutual exclusion algorithms. When two processes access a filter, it chooses one of two processes to be a winner. In their algorithms, the number of filters that a process has to access in order to acquire a lock depends on the degree of contention. The higher the contention, the more filters that have to be accessed. In this way, their algorithm adapts to the level of contention.

Through adaptivity, Yang and Anderson, and Singh and Choy were able to improve upon the time complexity of previously known mutual exclusion algorithms within the constraint of atomic reads and writes. However, since reducing time complexity was the objective, the research largely ignored the constant factors involved in using adaptivity. Furthermore, although an improvement over previous algorithms, their adaptive algorithms are still inferior to mutual exclusion algorithms that utilize atomic read-modify-write primitives.

This thesis considers the best known algorithms that take advantage of atomic read-modify-write primitives, and presents algorithms that improve upon them through adaptivity. From a practical standpoint, this approach is more relevant since almost all current shared-memory systems provide atomic read-modify-write primitives. This thesis is also concerned with the added run-time costs for dynamically selecting protocols.

Instead of crafting protocol-specific methods for selecting between locking protocols, this thesis provides generic protocol selection algorithm that can be used to select among any locking protocols.

Adaptive Barriers The most scalable algorithms for barriers rely on a software combining tree to achieve $O(logN)$ barrier latency. However, if barrier arrival times are skewed, the use of a combining tree to accumulate barrier arrivals leads to a higher latency than a simple, centralized counter. This observation led Gupta and Hill [20] to propose an *adaptive combining tree* barrier that adapts the shape of the combining tree to the arrival patterns of the participating processes. They show that their algorithm leads to improved time complexities. However, their analysis of the algorithm ignores the run-time overhead of reconfiguring the combining tree.

In later work [49], Scott and Mellor-Crummey investigated the performance of Gupta and Hill's adaptive combining-tree barriers and found that the adaptive combining tree fails to outperform conventional tree and dissemination barriers. If processes arrive simultaneously at a barrier, a non-adaptive combining tree barrier will perform better because it does not have to pay the run-time overhead of adaptivity. If processes arrive skewed in time, the length of time in between barrier episodes will be sufficiently long that the reduction in latency for detecting the last process is insignificant.

The adaptive combining tree does however show a performance advantage when used as a *fuzzy barrier* [20]. In a fuzzy barrier, a process waiting at the barrier can perform some useful computation that does not rely on completion of the barrier. This shows that the main advantage of the adaptive combining tree barriers comes from allowing the waiting processes to perform more useful work while waiting.

5.2 Complementary Methods

Besides the approach in this thesis of designing efficient algorithms for synchronization operations, there exist other complementary methods for reducing synchronization costs. These methods can be used together with reactive synchronization algorithms for reducing synchronization costs.

5.2.1 Program Restructuring

This approach restructures the synchronization pattern of a program to minimize data-dependencies and avoid any unnecessary synchronization delays. For example, barrier synchronization is frequently used to enforce data-dependencies across phases of a program. However, barrier synchronization presents two major drawbacks: it requires global communication and it unnecessarily delays computation. Instead of barriers, programs can use data-level or point-to-point synchronization to enforce data dependencies.

Kranz *et al.* [28] and Yeung and Agarwal [56] investigated the performance benefits of restructuring a program to use fine-grained synchronization. They also investigated the benefits of providing hardware support for efficient data-level synchronization. They found that restructuring the program to use fine-grained synchronization instead of barriers improves performance by a factor of three due to increased parallelism. Hardware support for fine-grained, data-level synchronization in the form of full/empty bits [53] yields an additional 40% performance improvement.

Nguyen [46] used compiler analysis to transform statically partitioned DOALL loops to use point-to-point communication between processors instead of global barriers. Conventional implementations of DOALL loops use a barrier at the end of each DOALL loop to enforce data-dependencies across DOALL loops. However, barriers enforce unnecessary dependencies across all the processors. To avoid over-constraining the processors, compiler analysis identifies the essential inter-processor dependencies and enforces them with point-to-point synchronization operations instead of barriers. Experimental results show about a factor of two improvement in execution times.

An effect of restructuring programs to synchronize at a finer granularity is to increase the frequency of synchronization operations, while reducing contention at each synchronization operation and shortening waiting times. This makes the right choice of protocols and waiting mechanisms even more important, and further motivates the need for reactive synchronization algorithms.

5.2.2 Multithreading

Multithreading is commonly prescribed as a method for tolerating latencies and increasing processor utilization in a large-scale multiprocessor. It accomplishes this by rapidly switching the processor to a different thread whenever a high-latency operation is encountered. While previous multithreaded designs switch contexts at every cycle [53, 21], Alewife's multithreaded processor [1] switches contexts only on synchronization faults and remote cache misses. This style is called *block multithreading* [33] and has the advantage of high single thread performance. In this thesis, we considered multithreaded processors as providing additional waiting mechanisms to be selected by a waiting algorithm.

5.3 Concurrent Search Structure Algorithms

A search structure algorithm implements the dictionary abstract data type. In [51], Shasha and Goodman present a framework for designing and verifying concurrent search structure algorithms. They exploit the semantics of the dictionary abstract data type to design and verify highly concurrent search structure algorithms. In their model, a search structure algorithm stores dictionary entries in the nodes of a graph, and dictionary *member*, *insert* and *delete* operations traverse the graph and manipulate the graph nodes to perform the operation.

A concurrent search structure algorithm avoids locking in order to increase concurrency. Shasha and Goodman propose a *give-up* technique that achieves this objective. In this technique, an process may arrive at a graph node, expecting to find a particular member in that node. However, another concurrent dictionary operation may violate that expectation. If this happens, the algorithm simply gives up and retries the search. Shasha and Goodman observe that simulation studies show that the give-up technique results in better performance than techniques that another technique that requires more locking.

We observe that the give-up technique is similar to our technique of serializing protocol changes with consensus objects. As in the give-up technique, our technique also attempts to permit more concurrency by reducing locking requirements and allowing the algorithm to execute an invalid protocol. An invalid protocol execution causes the reactive algorithm to give-up and retry the synchronization operation.

Chapter 6

Summary and Future Work

6.1 Summary

This thesis explores the performance implications of using run-time information to enhance the performance of synchronization operations. It first identifies the potential benefits of selecting protocols and waiting mechanisms based on contention and waiting times. However, in order to realize the potential benefits, we have to minimize the run-time cost of making the choice. This thesis designs reactive synchronization algorithms that perform the run-time selection with minimal overhead, given reasonable assumptions about the run-time behavior of parallel applications.

Protocol Selection Algorithms The first part of this thesis deals with the problem of selecting protocols correctly and efficiently. It presents a framework for reasoning about and designing efficient protocol selection algorithms with minimal run-time cost. We assume that run-time contention levels do not vary in such a way as to require frequent protocol changes, and optimize for the case when the currently selected protocol is optimal. We permit the maximum amount of concurrency in the presence of dynamic protocol changes by optimistically executing protocols and detecting later if the protocol was the correct protocol to use.

The design framework shows how minor modifications to a synchronization protocol allows it to be dynamically disabled and enabled. We defined$\mathcal{C}$-serializability as a correctness condition and introduced the notion of consensus objects as a method of satisfying $\mathcal{C}$-serializability. We also presented a 3-competitive policy for deciding when to change synchronization protocols.

Using the framework, we designed and implemented reactive algorithms for spin locks and fetch-and-op that are based on consensus objects for correctness. These algorithms require minor modifications to the original protocols and are implemented in C. Since the reactive algorithms rely only on the existence of shared-memory read-modify-write operations, they will run on any platform that supports a shared-memory abstraction. Experimental results on both a simulated and real Alewife multiprocessor demonstrate that the reactive spin lock and fetch-and-op algorithms can approach and even outperform the performance of conventional passive algorithms.

This thesis shows how a protocol selection algorithm can dynamically select between shared-memory and message-passing protocols. The advantage of using message-passing is that it is typically more efficient than using shared-memory under high contention. Unfortunately, the fixed overheads of message sends and receives make message-passing protocols more expensive than shared-memory protocols under low contention. Reactive algorithms provide a solution by deferring the choice of protocols to run-time.

The reactive spin lock algorithm removes the need for special hardware support for queue locks. For example, the Stanford DASH multiprocessor [38] and the Wisconsin Multicube [15] both include hardware support for queuing lock waiters. Software queuing algorithms provide the same scalable performance as hardware queue locks, but they come at a price of higher lock latency in the absence of contention. Our reactive spin lock solves the latency problem, thus eliminating the incentive of providing queuing in hardware.

The reactive fetch-and-op algorithm provides a viable alternative to hardware combining networks. For example, the NYU Ultracomputer [17] includes combining in its interconnection network. While software combining algorithms offer an alternative, they come at a price of extremely high latency at low contention levels. Our reactive fetch-and-op algorithm solves the latency problem at low contention levels and provides scalable throughput. Although a hardware combining network will result in higher throughput, its additional complexity and cost will have to be justified against a reactive fetch-and-op algorithm.

Waiting Algorithms The second part of this thesis deals with the problem of dynamically selecting waiting mechanisms. This thesis divides waiting mechanisms into two fundamental classes: polling and signaling mechanisms. The waiting cost of polling is proportional to the waiting time, while the waiting cost of signaling is a fixed cost, B. Previous research designed two-phase waiting algorithms that poll until the cost of polling reaches a limit, L_{poll}, before resorting to a signaling mechanism. Previous research also designed methods of choosing L_{poll} dynamically so as to achieve an optimal on-line competitive factor of 1.58. However, they incur a significant run-time overhead in deciding L_{poll} dynamically.

This thesis attempts to minimize the run-time cost of deciding L_{poll} by relying on waiting times to follow some probability distribution. We argue that exponentially and uniformly distributed waiting times are common, assuming Poisson arrivals of synchronizing threads. Experimental measurements corroborate this hypothesis. We derived theoretical results that show that under exponentially distributed waiting times, a static choice of $L_{poll} = 0.54B$ yields a 1.58-competitive waiting algorithm, and that under uniformly distributed waiting times, a static choice of $L_{poll} = 0.62B$ yields a 1.62-competitive waiting algorithm. These competitive factors are very close to the theoretical minimum for on-line waiting algorithms. In practice, these results indicate that a static two-phase waiting algorithm should poll for about half the cost of signaling rather than the entire cost of signaling.

Measurements of parallel applications confirm that two-phase waiting algorithms are very robust, and that polling for about half the cost of signaling can yield improved performance. Interestingly, the results also show that blocking is a good waiting mechanism in Alewife. This is due to low blocking overheads in Alewife's streamlined and minimal run-time thread management system. This fact re-emphasizes the importance of minimizing the cost of blocking. When the cost of blocking is comparable to waiting times, any reduction in the cost of blocking will immediately show up as a reduction in the cost of waiting.

6.2 Future Work

This thesis demonstrates the possibility of dynamically selecting protocols and waiting mechanisms to improve the performance of synchronization algorithms. It would be interesting to see if the techniques developed in this thesis can be applied to the design of reactive algorithms for other synchronization operations that exhibit a run-time choice between protocols. Other synchronization operations that may benefit from dynamic protocol selection are reader-writer locks and barriers.

This thesis also raises interesting issues for future research. In the rest of this section, we outline several areas that may be worth investigating.

Combining Protocol Selection and Waiting Mechanism Selection

The reactive algorithms in this thesis select protocols and waiting mechanisms separately. This ignores an opportunity to perform the run-time selection at the same time. The rationale behind this is the observation that contention levels and waiting times are usually correlated. Thus, run-time conditions that favor a particular protocol may also favor a particular waiting mechanism. For example, in designing the reactive spin lock, a better approach may be to use a polling mechanism for the test-and-test-and-set protocol, and a signaling mechanism for the queuing protocol.

Verifying Protocol Selection Algorithms

Chapter 3 defines $\mathcal{C}$-serializability as a correctness condition that an implementation should satisfy to allow it to be selected dynamically. It also introduces the notion of consensus objects as a means of satisfying it. There may be other means of satisfying $\mathcal{C}$-serializability, and it would be useful to further develop the theory to allow a designer to verify whether his/her protocol satisfies it.

A possible approach is to restrict our attention to *linearizable* implementations of concurrent objects [23]. Such objects can be specified using standard sequential axiomatic specifications. We can extend a sequential specification of a linearizable object in the following way. We add a term to the precondition of each axiom in the original specification that states that if a protocol's object is invalid, then the operation on that object should return an exception value. We then specify two new operations to validate and invalidate the object. An implementation of this extended specification would need to serialize protocol changes, thus satisfying $\mathcal{C}$-serializability. An algorithm designer can then use the methodology developed in [23] to determine if an implementation satisfies the extended specification.

Policies for Switching Protocols

Recall that a policy for deciding when to switch protocols first needs to monitor run-time conditions to decide which protocol is optimal for the current conditions. If it finds that the protocol in use is not optimal, it must then decide whether to switch to the optimal protocol.

We currently require the designer of a protocol selection algorithm to profile the execution of each protocol to determine which protocol is optimal for a given level of contention. Although this process needs to be performed only once for each machine architecture, it would be interesting to see if it can be automated. One approach is to model the performance of each protocol in terms of some relevant architectural parameters so that the tradeoffs between different protocols can be easily predicted for a given architecture [10].

It may also be possible to design more sophisticated policies for deciding when to switch protocols. The primary goal is to defend against the possibility of thrashing between protocols. In this thesis, we explored using hysteresis and competitive techniques for deciding when to switch protocols. These policies essentially set some thresholds for using a sub-optimal protocol before switching to another protocol. A possible extension to these policies would be to detect thrashing and dynamically adjust the thresholds accordingly. The switching thresholds should be increased when thrashing is detected and decreased otherwise. The objective is to adapt the policy to the evolution of run-time conditions.

Feedback to a Compiler or Programmer

Contention levels and waiting times in a parallel program may be hard to predict by statically analyzing the program text. However, for programs where contention and waiting times at a synchronization object remain consistent across multiple program runs, it may be worth the effort to profile the run-time behavior of the synchronization objects and report the results to a compiler or a programmer. For example, while analyzing the results of protocol selection algorithms in Section 3.5, we identified two types of locks used in MP3D: a low-contention lock for updating cell parameters, and a high-contention lock for updating collision counts. Appropriate feedback may allow the programmer or compiler to fix the choice of protocols for these two different types of locks in MP3D.

Appendix A

An Overview of Spec

This appendix gives a brief overview of the Spec language [36] that is used in Appendix B for specifying and describing several implementations of protocol selection algorithms.

Spec is a language designed by Butler Lampson and William Weihl for writing specifications and the first few stages of successive refinement towards practical implementations of digital systems, all in a common syntax. Spec provides a succinct notation for writing precise descriptions of sequential or concurrent systems, both sequential and concurrent. It is essentially a notation for describing allowable sequences of transitions of a state machine. A complete description of Spec's syntax and semantics is presented in [36].

This purpose of this overview is to aid the reader in understanding the Spec code. We concentrate on the features of Spec that are different from, or absent from conventional programming languages. The overview is largely derived from the handouts describing Spec in [36].

Expressions and Commands

The Spec language has two main constructs: an *expression* that describes how to compute a value as a function of other values (literal constants, or current values of state variables) without any side-effects, and a *command* that describes possible transitions of the state variables. They loosely correspond to expressions and statements in a conventional language. Spec expressions are *atomic*, while commands can be atomic or non-atomic.

An atomic command is specified using *atomicity brackets* `<<` and `>>`. A non-atomic command is a sequence of atomic commands that can be interleaved with some other concurrent computation.

Program Organization

In addition to expressions and commands, Spec has three constructs that are useful for organizing a program.

- A *routine*, which is an abstraction of a piece of computation. There are three kinds of routines: 1) a *function* (defined with `FUNC`) that is an abstraction of an expression, 2) an *atomic procedure* (defined with `APROC`) that is an abstraction of an atomic command, and 3) a *general procedure* (defined with `PROC`) that is an abstraction of a non-atomic command.
- A *type*, which is an assertion about the set of values that a name can assume. Spec includes the most of the standard types of a conventional language, and also includes *sets* and *sequences* as additional built-in types to ease specification.
- A *module*, that structures the name space into a two-level hierarchy. An identifier `i` declared in a module `m` has the name `m.i` throughout the program.

A Spec program is organized as a set of modules and some global declarations. Each module defines a number of types, variables, and routines.

Type Naming Convention
Spec is strongly typed, and requires the user to declare the types of all variables, just as in Pascal. There is a convention, however, that allows a user to omit explicit type declarations. If `Foo` is a type, it can be omitted in a declaration of the variables `foo`, `foo1`, `foo'`, etc. That is, the type of a variable whose type has not been explicitly declared is derived by dropping all trailing digits and `'`s from the name and using the type with the same name except for capitalization.

Expression Operators
Spec expressions include the standard arithmetic and logical operators for combining expressions into larger ones. Spec uses mostly conventional symbols to denote them. However, it uses unconventional symbols for the following operators. /\ denotes conditional "and", and \/ denotes conditional "or". `#` denotes "not equal".

Quantifiers
Spec has existential and universal quantifiers (`ALL` and `EXISTS`) that make it easy to describe properties without explicitly stating how to test for them in a practical way. For instance, the following expression is true iff the sequence `s` is sorted:

```
(ALL i : INT | 0 <= i /\ i < s.size-1 ==> s[i] <= s[i+1])
```

The expression is read as, "for all `i` such that $0 \leq \texttt{i} < \texttt{s.size} - 1, \texttt{s[i]} \leq \texttt{s[i+1]}$". The `==>` symbol is logical implication.

Pointers and Dereferencing

If `x` is an object of type `T`, then `x.new` returns a pointer to that object. The returned pointer has type `REF[T]`. `p^` dereferences pointer `p` and returns the object pointed to by `p`.

Command Operators

Spec has several operators for combining primitive commands into larger ones. The main primitive commands are assignment and routine invocation. There are also primitive commands to return a result from a routine (`RET`) and to do nothing (`SKIP`). The operators used in this thesis are:

- A conditional operator: `a => b`, which is read as "if `a` then `b`". `a` is called the guard of the command. If `a` is false, the command *fails* and simply waits until `a` becomes true sometime in the future. Contrast this with "if" statements in conventional languages that continues execution of the next statement if the predicate fails.

- Choice operators: `c1 [] c2` and `c1 [*] c2`. `c1 [] c2` makes a non-deterministic choice between `c1` and `c2`. It chooses one that doesn't fail. Usually `c1` and `c2` will be guarded with a conditional operator. `c1 [*] c2` executes `c1` unless `c1` fails, in which case it executes `c2`. Thus, one can read `[]` as "or", and `[*]` as "else".

  For example,

  ```
     x = 1  => y := 0
  [] x >= 1 => y := 1
  ```

  sets `y` to 1 if $\texttt{x} > 1$, non-deterministically sets `y` to 0 or 1 if $\texttt{x} = 1$, and does nothing if $\texttt{x} < 1$.

  Also,

  ```
      x = 1  => y := 0
  [*] x >= 1 => y := 1
  ```

  sets `y` to 1 if $\texttt{x} > 1$, sets `y` to 0 if $\texttt{x} = 1$, and does nothing if $\texttt{x} < 1$.

- A sequencing operator: `c1; c2`, which means to execute `c1` followed by `c2`.

- A looping operator: `DO command OD`, which means to execute `command` until it fails. The most common use is `DO P => Q OD`, which is read as "while `P` is true do `Q`".

- Variable introduction: `VAR id | command`, which means to choose a variable `id` such that `command`. The most common use is the form `VAR x:T | P(x) => Q`, which is read as "choose some `x` of type `T` such that `P(x)`, and do Q". It fails if there is no `x` for which `P(x)` is true.

Example

Here is an example specification for a procedure to search a sequence for a given element:

```
APROC Search (a:SEQ[INT], x:INT) -> UNION(INT, NULL) =
    << VAR i:INT | (0 <= i /\ i < a.size /\ a[i] = x) => RET i
       [*] RET nil >>
```

The specification says that the `Search` procedure should return *any* index `i` for which `a[i] = x`. If there is no such index, then `Search` should return `nil`. Here is an implementation of the above specification that returns the smallest index `i` for which `a[i] = x`, or `nil` if there is no such index.

```
APROC Search (a:SEQ[INT], x:INT) -> UNION(INT, NULL) =
    << VAR i:INT := 0 |
         DO
           i < a.size /\ a[i] # x => i := i+1
         OD;
         i = a.size => RET nil
     [*] RET i
    >>
```

Appendix B

Specification of Protocol Selection Algorithms

This appendix supplements the description of the framework for designing and reasoning about protocol selection algorithms presented in Chapter 3 by providing specifications and implementations of protocol objects and managers in the Spec language. The motivation for using Spec is that it provides a more precise and succinct description than the pseudo-code in Chapter 3. Obviously, the primary drawback of using Spec is that it requires the reader to understand the semantics of Spec. Appendix A gives a brief overview of the Spec language.

B.1 A Sequential Specification of a Protocol Object

Figure B.1 presents the sequential specification of a protocol object. This corresponds to a sequential execution of the specification in Figure 3.5. The specification exhibits only sequential behavior because each of the procedures in the specification are atomic (`APROC`).

`DoProtocol` returns the result of executing the protocol associated with the protocol object if `p.valid` is `true`. Otherwise it returns `nil`. We assume that `nil` is not one of the return values of `P.RunProtocol`.

`Invalidate` changes `p.valid` to `false` and returns `true` if `p.valid` is `true`. Otherwise it returns `false`.

`Validate` assumes that `p.valid` is false when it is called. Otherwise, its behavior is undefined. It resets the protocol to a consistent state by calling `P.UpdateProtocol` and sets `p.valid` to true.

`IsValid` simply returns the value of `p.valid`.

```
MODULE ProtocolObject[P, V] =

APROC DoProtocol(p) -> UNION[V, NULL] =
   << p.valid => VAR v := P.RunProtocol(p) | RET v
  [*] RET nil >>

APROC Invalidate(p) -> BOOL =
   << p.valid => p.valid := false; RET true
  [*] RET false >>

APROC Validate(p) =
   << p.valid => HAVOC
  [*] P.UpdateProtocol(p); p.valid := true >>

FUNC IsValid(p) -> BOOL = RET p.valid

END ProtocolObject
```

Figure B.1: *A sequential specification of a protocol object.*

B.2 A $\mathcal{C}$-serial Specification of a Protocol Object

Figure B.2 presents a $\mathcal{C}$-serial specification of a protocol object. This specification provides an alternative method of defining of a $\mathcal{C}$-serial execution, without the use of histories. The specification allows only executions where protocol changes are serialized with respect to other operations by detecting concurrent protocol changes. A concurrent protocol change operation indicates a violation of a $\mathcal{C}$-serial execution and causes all further protocol executions and changes to block. This specification is equivalent to a $\mathcal{C}$-serial execution of the specification in Figure 3.5

Unlike the sequential specification, this specification does not use atomic procedures, allowing the procedures to be executed concurrently. The specification ensures $\mathcal{C}$-serial executions by calling `Begin` and `End` at the beginning and end of each operation, respectively. It uses unique id's and a map of unique id's to operation types to keep track of the type of each operation in progress. `Begin` adds an operation to the set of pending operations in `pending`. `End` checks if a protocol change is concurrent with another operation by calling the function `CSerial`. `CSerial` returns `false` if a protocol change is concurrent with any other operation. `End` blocks if `CSerial` returns `false`.

```
MODULE ProtocolObject[P, V, ID] =

TYPE OPTYPE = UNION[EXEC, CHANGE]      % operation type
     Y      = ID -> OPTYPE             % map of op ids to op types

VAR  y0      := Y{}                    % initial map
     pending : SET[ID] := {}           % pending operations

APROC Begin(optype) -> ID =
  << VAR id | ~ id IN pending =>
        y0(id) := optype;
        pending := pending ++ id;
        RET id >>

APROC End(id, v) =  % blocks if another change op is concurrent
  << CSerial(y0, id) => pending := pending -- id; >>

FUNC Cserial(y, id) -> BOOL =
  RET (ALL id' in pending |
           y(id) = CHANGE ==> id = id' /\
           y(id) = EXEC ==> y(id') = EXEC )

% Interface procedures
PROC DoProtocol(p) -> UNION[V, NULL] =
    VAR v : UNION[V, NULL] := nil,
        id := Begin(EXEC) |
        BEGIN p.valid => v := P.RunProtocol(p) [*] SKIP END;
        End(id, EXEC);
        RET v

PROC Invalidate(p) -> BOOL =
    VAR b : BOOL := false,
        id := Begin(CHANGE) |
        BEGIN p.valid => p.valid := false; b := true [*] SKIP END;
        End(id, CHANGE);
        RET b

PROC Validate(p) =
    p.valid => HAVOC
[*] VAR id := Begin(CHANGE) |
        P.UpdateProtocol(p);
        p.valid := true;
        End(id, CHANGE)

FUNC IsValid(p) -> BOOL = RET p.valid
```

Figure B.2: *A $\mathcal{C}$-serial specification of a protocol object.*

```
MODULE ProtocolManager[P1, P2, V] =

TYPE PS = SEQ[PROTOCOL_OBJECT]

PROC Create() -> PS =
    VAR ps := PS{ P1.Create(), P2.Create() } |
        Invalidate(ps[1]);
        RET p

PROC DoSynchOp(ps) -> V =
    VAR v : UNION[V, NULL] := nil |
        DO v = nil =>
            v := DoProtocol(ps[0])
         [] v := DoProtocol(ps[1])
        OD;
        RET v

PROC DoChange(ps) =
    Invalidate(ps[0]) => Validate(ps[1])
 [] Invalidate(ps[1]) => Validate(ps[0])
```

Figure B.3: *A protocol manager.*

B.3 An Implementation of a Protocol Manager

Figure B.3 provides essentially the same implementation of a protocol manager as in Figure 3.6, except that it is in Spec. `DoSynchOp` non-deterministically chooses one of the protocols to execute until it succeeds in executing a valid protocol. It returns the result of the valid execution. `DoChange` validates a protocol only if it succeeds in invalidating a valid protocol.

B.4 A $\mathcal{C}$-serializable Implementation of a Protocol Object

As observed in Chapter 3 we canonically describe the execution of a protocol with consensus objects as such:

```
MODULE ProtocolObject[P, V] =

PROC DoProtocol(p) -> UNION(V, NULL) =
    VAR v |
        P.PreConsensus(p) =>
          BEGIN
            P.AcquireConsensus(p);
            P.InConsensus(p);
            p^.valid => P.ReleaseConsensus(p);
                        RET P.PostConsensus(p);
        [*] P.ReleaseConsensus(p);
            P.PostConsensusFail(p); RET nil
          END
    [*] P.WaitConsensus(p) => RET P.PostConsensus(p);
    [*] P.PostConsensusFail(p); RET nil

PROC Validate(p) =
    P.AcquireConsensus(p);
    p^.valid => HAVOC % should not happen
[*] P.Update(p);
    p^.valid := true;
    P.ReleaseConsensus(p)

PROC Invalidate(p) -> BOOL =
    P.AcquireConsensus(p);
    p^.valid =>
        p^.valid := false;
        P.ReleaseConsensus(p);
        RET true
[*] P.ReleaseConsensus(p);
    RET false

FUNC IsValid(p) -> BOOL =
    RET p^.valid

END ProtocolObject
```

Figure B.4: *A $\mathcal{C}$-serializable implementation of a protocol object based on consensus objects.*

```
PROC RunProtocol(p) =
    PreConsensus(p) =>            % pre-consensus phase
        AcquireConsensus(p);
        InConsensus(p);           % in-consensus phase
        ReleaseConsensus(p);
        PostConsensus(p)
[*] WaitConsensus(p);             % wait-consensus phase
    PostConsensus(p)              % post-consensus phase
```

This structure allows us to implement $\mathcal{C}$-serializable protocol objects without the explicit use of locks. Figure B.4 provides an implementation of a protocol object that relies on the atomicity provided by consensus objects to ensure that protocol changes are $\mathcal{C}$-serializable.

This implementation corresponds to the pseudo-code in Figure 3.11.

B.5 A Generic Protocol Selection Algorithm for Lock Protocols

As a concrete example, we use the design framework to implement generic protocol selection algorithms for mutual-exclusion locks (mutexes) and reader-writer locks. Locking protocols trivially satisfy the property of consensus objects: we use the locks themselves as the consensus objects. `PreConsensus()`, `InConsensus()`, `WaitConsensus()` and `PostConsensus()` are null functions. For mutexes, `AcquireConsensus()` is equivalent to acquiring the mutex, and `ReleaseConsensus()` is equivalent to releasing the mutex. For reader-writer locks, `AcquireConsensus()` is equivalent to acquiring a write-lock, and `ReleaseConsensus()` is equivalent to releasing a write-lock.

Figure B.5 presents an implementation of a protocol manager and a protocol object for mutual-exclusion lock protocols. In the protocol manager, `DoSynchOp` is split into `Acquire` and `Release`. We omit the type declarations and the definition of `Create`: they are identical to the ones in Figure B.3. In the protocol object, `P.Lock` and `P.Unlock` are the original mutex protocols for acquiring and releasing a lock, respectively. Thus, one can simply plug in any existing mutex protocol in this template to get an initial design of a protocol selection algorithm for mutex protocols.

Similarly, Figure B.6 presents an implementation of a protocol manager and a protocol object for reader-writer lock protocols. `DoSynchOp` is split into `AcquireRead`, `AcquireWrite`, `ReleaseRead`, and `ReleaseWrite`.
`P.ReadLock`, `P.ReadUnlock`, `P.WriteLock`, and `P.WriteUnlock` are the original reader-writer lock protocols.

```
MODULE MutexManager[P1, P2] =

PROC Acquire(ps) =
    VAR b : BOOL := fail |
        DO b = fail =>
            IsValid(ps[0]) => b := Acquire(ps[0])
         [] IsValid(ps[1]) => b := Acquire(ps[1])
        OD

PROC Release(ps) =
    IsValid(ps[0]) => Release(ps[0])
 [] IsValid(ps[1]) => Release(ps[1])


PROC DoChange(ps) =
    Invalidate(ps[0]) => Validate(ps[1])
 [] Invalidate(ps[1]) => Validate(ps[0])

END MutexManager


MODULE MutexObject[P] =

PROC Acquire(p) -> BOOL =
    P.Lock(p);
    p^.valid => RET success
[*] P.Unlock(p);
    RET fail

PROC Validate(p) =
    P.Lock(p);
    p^.valid => HAVOC
[*] p^.valid := true;
    P.Unlock(p)

FUNC IsValid(p) -> BOOL =
    RET p^.valid

PROC Release(p) =
    P.Unlock(p)

PROC Invalidate(p) -> BOOL =
    P.Lock(p);
    p^.valid =>
        p^.valid := false;
        P.Unlock(p);
        RET true
[*] P.Unlock(p);
    RET false

END MutexObject
```

Figure B.5: *A protocol selection algorithm for mutual-exclusion locks, based on consensus objects.*

```
MODULE RWLockManager[P1, P2] =

PROC AcquireRead(ps) =
  VAR b : BOOL := fail |
    DO b = fail =>
      IsValid(ps[0]) =>
        b := AcquireRead(ps[0])
   [] IsValid(ps[1]) =>
        b := AcquireRead(ps[1]) OD

PROC AcquireWrite(ps) =
  VAR b : BOOL := fail |
    DO b = fail =>
      IsValid(ps[0]) =>
        b := AcquireWrite(ps[0])
   [] IsValid(ps[1]) =>
        b := AcquireWrite(ps[1]) OD

PROC ReleaseRead(ps) =
    IsValid(ps[0]) =>
      ReleaseRead(ps[0])
 [] IsValid(ps[1]) =>
      ReleaseRead(ps[1])

PROC ReleaseWrite(ps) =
    IsValid(ps[0]) =>
      ReleaseWrite(ps[0])
 [] IsValid(ps[1]) =>
      ReleaseWrite(ps[1])

PROC DoChange(ps) =
    Invalidate(ps[0]) => Validate(ps[1])
 [] Invalidate(ps[1]) => Validate(ps[0])

END RWLockManager

MODULE RWLockObject[P] =

PROC AcquireRead(p) -> BOOL =
    P.ReadLock(p);
    p^.valid => RET success
[*] P.ReadUnlock(p);
    RET fail

PROC AcquireWrite(p) -> BOOL =
    P.WriteLock(p);
    p^.valid => RET success
[*] P.WriteUnlock(p);
    RET fail

PROC ReleaseRead(p) =
    P.ReadUnlock(p)

PROC ReleaseWrite(p) =
    P.WriteUnlock(p)

PROC Validate(p) =
    P.WriteLock(p);
    p^.valid => HAVOC
[*] p^.valid := true;
    P.WriteUnlock(p)

FUNC IsValid(p) -> BOOL =
    RET p^.valid

PROC Invalidate(p) -> BOOL =
    P.WriteLock(p);
    p^.valid =>
        p^.valid := false;
        P.WriteUnlock(p);
        RET true
[*] P.WriteUnlock(p);
    RET false

END RWLockObject
```

Figure B.6: *A protocol selection algorithm for reader-writer locks, based on consensus objects.*

Appendix C

The Reactive Fetch-and-Op Algorithm

This appendix presents the pseudo-code for our reactive fetch-and-op algorithm. We begin by describing Goodman *et al.*'s combining tree algorithm, one of the protocols selected by the reactive fetch-and-op algorithm, before presenting the pseudo-code for the reactive fetch-and-op.

C.1 Goodman *et al.*'s Combining Tree Algorithm

Figures C.1 and C.2 present the pseudo-code for Goodman *et al.*'s algorithm. The algorithm computes fetch-and-add, although it can be easily modified to compute any associative operation. The value of the fetch-and-op operation is stored in the root of the combining tree, and processes traverse the tree in order to update the value.

The algorithm consists of three phases. In the first phase, a process moves up the tree "claiming ownership" of each visited node until it reaches a node that has been claimed by some other process. Call this node the process' final node. In the second phase, the process revisits the nodes it has claimed, combining operations with later arrivals at those nodes along the way, and posting its combined value at the final node. In the third phase, the process waits for the owner of that final node to post its result. The process then descends the tree, distributing its result to waiters at the nodes it owns.

In the pseudo-code, Parts One and Two correspond to the first and second phase, respectively, while Parts Three and Four correspond to the third phase. Each node in the tree consists of six fields: *status*, *wait*, *first_incr*, *second_incr*, *result*, *parent* and *children*. The *wait* field indicates if a process is in phase three and waiting for a result at that node. *first_incr* stores the combined value of the subtree visited by the owner of the node. *second_incr* stores the combined value of the subtree visited by the waiting process (waiting

```
 type node = record  // combining tree node
     status : (FREE, COMBINE, RESULT)
     wait   : boolean
     first_incr : integer
     second_incr : integer
     result : integer
     parent : ^node
     children : ^node


 procedure fetch_and_add(counter : tree, value : integer) returns integer
     saved_result : integer
     leaf : ^node := get_leaf(counter, pid)
     node : ^node := leaf

// Part One, find path up to first COMBINE or ROOT node (pre-consensus)
     going_up : boolean := TRUE
     repeat while going_up
         lock(node)
         if node->status = RESULT
             unlock(node)
             repeat while node->status = RESULT
         else if node->status = FREE
             node->status := COMBINE
             unlock(node)
             node := node->parent
         else // COMBINE or ROOT node
             going_up := FALSE

// Part Two, lock path from Part 1, combining values along the way (pre-consensus)
     total : integer := value
     visited : ^node := leaf
     repeat while visited != node
         lock(visited)
         visited->first_incr := total
         if visited->wait // do combining
             total := total + visited->second_incr
         visited := visited->parent
```

Figure C.1: *Goodman et al.'s Combining Tree: Parts One and Two.*

```
// Part Three, operate on last visited node (in-consensus or wait-consensus)
     if node->status = COMBINE
         node->second_incr := total
         node->wait := TRUE
         repeat while node->status = COMBINE
             unlock(node)
             repeat while node->status = COMBINE
             lock (node)
         node->wait := FALSE
         saved_result := node->result
         node->status := FREE
     else
         saved_result := node->result
         node->result := result + total

// Part Four, descend tree and distribute results (post-consensus)
     unlock(node)
     repeat until is_leaf_node(node)
         node  := get_child(node, pid)
         if node->wait
             node->status := RESULT
             node->result := saved_result + node->first_incr
         else
             node->status := FREE
         unlock(node)

     return saved_result
```

Figure C.2: *Goodman et al.'s combining tree: Parts Three and Four.*

in phase three) at that node. *result* communicates to a waiting process the result to be distributed down the subtree visited by that process. Finally *parent* and *children* point to the parent and children of the node.

The following is a modified excerpt from [15] describing their algorithm. In Part One, a process progresses up the combining tree, marking each FREE node as a COMBINE node. If the process finds a RESULT node, it must wait until the node becomes either FREE or COMBINE before continuing up the tree. When a ROOT or COMBINE node is reached, that final node is locked, and the algorithm continues to Part Two.

In Part Two, the process locks each node previously visited, bottom up, and tallies the nodes' *second_incr* values. Along the way, the tally for the previous subtree is stored in *first_incr*. The revisited nodes remain locked until Part Four when results are distributed.

In Part Three, if the final node was a COMBINE node, then the final tally is added to *second_incr* for that node, the *wait* field is set to true, and the process spin waits until the node becomes a RESULT node. If the final node was a ROOT node, the *result* field is incremented by the total tally, essentially performing the fetch-and-add on the counter.

In Part Four, the process reverses its path down the tree, distributing results along the way. At each node, if there is a waiting process, the node's *result* field is set to the *result* from Part Three, plus its own subtree's increment *first_incr*, and the node's *status* is set to RESULT. Otherwise the node is re-initialized to FREE.

C.2 The Reactive Fetch-and-Op

Figures C.3–C.7 present the pseudo-code for our reactive fetch-and-op algorithm. The reactive algorithm computes fetch-and-add, although it can be easily modified to compute any associative operation. It is composed of a test-and-test-and-set lock-based counter, an MCS queue lock-based counter, and Goodman *et al.*'s software combining tree counter. Figure C.3 presents the data structures and the top-level dispatch code. The data structure is composed of the data structures of the component protocols and a mode variable. The mode variable indicates which of the three protocols is valid. As an optimization, the counter value associated with each protocol is kept in a common location.

The dispatch procedure, `fetch_and_add`, checks the mode variable to decide which protocol to use. Unlike the reactive lock, we cannot optimistically try the test-and-test-and-set lock-based counter since this will have the effect of serializing accesses to the combining tree, negating the benefits of parallelism under high contention.

Figures C.4–C.6 present the pseudo-code of the protocols being selected. `faa_tts`

```
type qnode = record
    next   : ^qnode
    status : (WAITING, GO, INVALID)

type node = record  // combining tree node
    status : (FREE, COMBINE, RESULT, ROOT, INVALID)
    wait   : boolean
    first_incr : integer
    second_incr : integer
    result : integer
    parent : ^node
    children : ^node

// The mode slot of the counter record should reside in a different
// cache line from the other slots
type counter = record
    mode       : (TTS, QUEUE, TREE)       // mode variable
    tts_lock   : (FREE, BUSY)             // slot for TTS lock
    queue_tail : (INVALID, ^qnode)        // slot for queue lock
    tree       : ^combining_tree          // slot for combining tree
    count      : integer                  // the fetch-and-op variable

procedure fetch_and_add (C : ^counter, value : integer) returns integer
    if L->mode = TTS
        return faa_tts (C, value)
    else if L->mode = QUEUE
        return faa_queue (C, value)
    else
        return faa_tree (C, value)
```

Figure C.3: *Reactive fetch-and-op: data structures and top-level dispatch code.*

implements the test-and-test-and-set lock-based counter, `faa_queue` implements the queue lock-based counter, and `faa_tree` implements the combining tree counter. Modifications to the original protocols are marked by “>” on the left end of each line. Additionally, “P>” denotes modifications for implementing the policy for changing protocols. As in the reactive spin lock, the original protocols have been modified to detect mode changes, and to abort and retry the synchronization operation upon detecting that the protocol is invalid. However, we omit the code for monitoring run-time conditions.

The policy for changing between the two lock-based protocols is similar to the one for the reactive spin lock. The policy for changing from the queue lock-based counter to the combining tree counter is based on queue waiting time. If the waiting time on the queue exceeds a limit for a small number of consecutive fetch-and-op operations. Since the queue

```
   procedure faa_tts (C : ^counter, value : integer) returns integer
       repeat while TRUE
           if C->tts_lock = FREE
               if test_and_set (&C->tts_lock) = FREE
                  count : integer := C->count
                  C->count := count + value
P>                if change_tts_to_queue_mode()  // in consensus, change protocols?
>                    tts_to_queue_mode(C)
>                 else
                     C->tts_lock := FREE
                  return count
           delay ()                              // do backoff
>          if C->mode != TTS
>              return fetch_and_add (C, value)

   procedure faa_queue (C : ^counter, value : integer) returns integer
       I : ^qnode := make_qnode()
       I->next := nil
       predecessor : ^qnode := fetch_and_store (&C->queue_tail, I)
       if predecessor != nil
>          if predecessor != INVALID             // queue was non-empty
               I->status := WAITING
               predecessor->next := I
               repeat while I->status = WAITING  // wait for GO or INVALID signal
>              if I->status = INVALID            // queue was invalid
>                  return fetch_and_add (C, value)
>          else                                  // queue was invalid
>              invalidate_queue (C, I)           // invalidate others on the queue
>              return fetch_and_add (C, value)
       count : integer := C->count               // do the add
       C->count := count + value
P>     if change_to_tts_mode()                   // in consensus, change protocols?
>          queue_to_tts_mode(C, I)
P>     else if change_to_tree_mode()
>          queue_to_tree_mode(C, I)
>      else
           release_queue(C, I)
       return count
```

Figure C.4: *Reactive fetch-and-op: test-and-test-and-set and queue lock-based protocols. Modifications to the original protocols are marked by "*`>`*". "*`P>`*" denotes modifications for implementing the policy for changing protocols.* `release_queue` *and* `invalidate_queue` *are identical to the ones defined in the pseudo-code for the reactive spin lock.*

```
 procedure faa_tree(C : ^counter, value : integer) returns integer
     saved_result : integer
     leaf : ^node := get_leaf(counter->tree, pid)
     node : ^node := leaf

// Part One, find path up to first COMBINE or ROOT node (pre-consensus)
     going_up : boolean := TRUE
     repeat while going_up
         lock(node)
         if node->status = RESULT
             unlock(node)
             repeat while node->status = RESULT
         else if node->status = FREE
             node->status := COMBINE
             unlock(node)
             node := node->parent
         else // COMBINE or ROOT node
             going_up := FALSE

// Part Two, lock path from Part 1, combining values along the way (pre-consensus)
     total : integer := value
     visited : ^node := leaf
     repeat while visited != node
         lock(visited)
         visited->first_incr := total
         if visited->wait   // do combining
             total := total + visited->second_incr
         visited := visited->parent
```

Figure C.5: *Reactive fetch-and-op: combining tree protocol, pre-consensus.*

```
  // Part Three, operate on last visited node (in-consensus or wait-consensus)
        if node->status = COMBINE               // wait-consensus
            node->second_incr := total
            node->wait := TRUE
            repeat while node->status = COMBINE
                unlock(node)
                repeat while node->status = COMBINE
                lock(node)
            node->wait := FALSE
>           if node->status = RESULT
>               saved_result := node->result
                node->status := FREE
>               part4(node, saved_result, RESULT)
                return saved_result
>           else  // node->status = INVALID, invalid root detected
>               node->status := FREE
>               part4(node, 0, INVALID)
>               return fetch_and_add(C, value)
>       else if node->status = ROOT             // in-consensus
            saved_result := C->count
            C->count := saved_result + total
P>          if change_tree_to_queue_mode()      // change protocols?
>               tree_to_queue_mode(C, node)
>           part4(node, saved_result, RESULT)
            return saved_result
>       else // node->status = INVALID, invalid root reached
>           part4(node, 0, INVALID)
>           return fetch_and_add(C, value)

  // Part Four, descend tree and distribute results (post-consensus)
   procedure part4(node : ^node, result : integer, status : integer)
        unlock(node)
        repeat until is_leaf_node(node)
            node := get_child(node, pid)
            if node->wait
                node->status := status
                node->result := result + node->first_incr
            else
                node->status := FREE
            unlock(node)
```

Figure C.6: *Reactive fetch-and-op: combining tree protocol, in-consensus, wait-consensus and post consensus. Modifications to the original protocol are marked by "`>`". "`P>`" denotes modifications for implementing the policy for changing protocols.*

is almost always FIFO, the waiting time is directly proportional to the level of contention at the counter. It is equal to the number of waiting processes multiplied by the time it takes to increment the counter and pass ownership of the queue lock to the next waiter. This has to be tuned for each different machine architecture.

The policy for changing from the combining tree to the queue lock-based counter is based on the number of combined requests reaching the root. The monitoring code, not shown in the pseudo-code, amounts to computing a fetch-and-increment along with the fetch-and-op, and seeing how large of an increment reaches the root. If the number of combined requests is below a threshold for some number of consecutive arrivals at the root, the algorithm initiates a switch back to the queue lock-based counter. Again, this has to be tuned for each machine architecture.

For the combining tree protocol, Parts One and Two correspond to the pre-consensus phase, Part Three corresponds to the in-consensus or wait-consensus phase, and Part Four corresponds to the post-consensus phase. In this protocol, a process that accesses an invalid root has a set of processes it combined with that are waiting for a return value. These waiting processes are in the wait-consensus phase. Thus, the process that reaches the invalid root completes the combining tree protocol by descending the combining tree and notifying the processes that it combined with to retry the fetch-and-op operation. This is implemented by setting `node->status` to `INVALID` in `phase4`.

Finally, Figure C.7 presents the pseudo-code for performing the mode changes. The names of the procedures are self-explanatory. In order to ensure that protocol changes are serializable with respect to other protocol executions and changes, these procedures are called only by processes that have successfully acquired a valid consensus object.

```
procedure tts_to_queue_mode(C : ^counter)
    I : ^qnode := make_qnode()
    acquire_invalid_queue(C, I)
    C->mode := QUEUE
    release_queue(C, I)

procedure queue_to_tts_mode(C : ^counter, I : ^qnode)
    C->mode := TTS
    C->tts_lock := FREE
    invalidate_queue(C, I)

procedure queue_to_tree_mode(C : ^counter, I : ^qnode)
    C->mode := TREE
    root : ^node := get_root(C)
    lock(root)
    root->status := ROOT
    unlock(root)
    invalidate_queue(C, I)

procedure tree_to_queue_mode(C : ^counter, root : ^node)
    // root is locked when called
    root->status := INVALID
    I : ^qnode := make_qnode()
    acquire_invalid_queue(C, I)
    C->mode := QUEUE
    release_queue(C, I)
```

Figure C.7: *Reactive fetch-and-op: making protocol changes. These routines are called only by processes that have acquired a valid consensus object.* `acquire_invalid_queue`, `release_queue` *and* `invalidate_queue` *are identical to the ones defined in the pseudo-code for the reactive spin lock.*

Bibliography

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 104–114, June 1990.

[2] Anant Agarwal and Mathews Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings 16th Annual International Symposium on Computer Architecture*, pages 396–406, New York, June 1989. IEEE. Also as MIT-LCS TM-396.

[3] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[4] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper appears as MIT/LCS Memo TM-454, 1991.

[5] Thomas E. Anderson. The Performance Implications of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, (Springer-Verlag Lecture Notes in Computer Science 279)*, pages 336–369, September/October 1986.

[7] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting Networks and Multi-Processor Coordination. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, pages 348–358, May 1991.

[8] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568, August 1991.

[9] Allan Borodin, Nathan Linial, and Michael Saks. An Optimal Online Algorithm for Metrical Task Systems. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 373–382, New York, May 1987. ACM.

[10] Eric A. Brewer. *Portable High-Performance Supercomputing: High-Level Architecture-Dependent Optimization*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1994.

[11] P.J. Burns *et al.* Vectorization of Monte-Carlo Particle Transport: An Architectural Study using the LANL Benchmark "Gamteb". In *Proc. Supercomputing '89*, New York, NY, November 1989. IEEE/ACM.

[12] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[13] Manhoi Choy and Ambuj K. Singh. Adaptive Solutions to the Mutual Exclusion Problem. In *12th Symposium on Principles of Distributed Computing (PODC)*, Ithaca, NY, 1993. ACM.

[14] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent Control with 'Readers' and 'Writers'. *Communications of the ACM*, 14(10):667–668, October 1971.

[15] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 64–75, April 1989.

[16] K. Gopinath, Krishna Narasimhan M. K., Beng-Hong Lim, and Anant Agarwal. Performance of Switch-Blocking in Multithreaded Processors. In *Proceedings of the 23rd International Conference on Parallel Processing*, Chicago, IL, August 1994.

[17] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing a MIMD Shared-Memory Parallel Machine. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[18] Allan Gottlieb, B. 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.

[19] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, pages 60–70, June 1990.

[20] Rajiv Gupta and Charles R. Hill. A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree. *International Journal of Parallel Programming*, 18(3):161–180, 1989.

[21] R.H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, New York, June 1988. IEEE.

[22] Robert H. Halstead. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[23] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. Technical Report CMU-CS-88-120, Carnegie-Mellon University, March 1988.

[24] Qin Huang. An Analysis of Concurrent Priority Queue Algorithms. Master's thesis, EECS Department, Massachusetts Institute of Technology, Cambridge, MA, August 1990.

[25] Anna Karlin, Kai Li, Mark Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. In *13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 41–55, October 1991.

[26] Anna Karlin, Mark Manasse, Lyle McGeoch, and Susan Owicki. Competitive Randomized Algorithms for Non-Uniform Problems. In *Proceedings 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 301–309, January 1990.

[27] Anna Karlin, Mark Manasse, Larry Rudolph, and Daniel Sleator. Competitive Snoopy Caching. *Algorithmica*, 3(1):79–119, 1988.

[28] David Kranz, Beng-Hong Lim, Anant Agarwal, and Donald Yeung. Low-cost Support for Fine-Grain Synchronization in Multiprocessors. In *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 7, pages 139–166. Kluwer Academic Publishers, 1994. Also available as MIT Laboratory for Computer Science TM-470, June 1992.

[29] David A. Kranz *et al.* ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of SIGPLAN '86, Symposium on Compiler Construction*, June 1986.

[30] Clyde 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.

[31] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *International Supercomputing Conference (ICS) 1993*, Tokyo, Japan, July 1993. IEEE.

[32] John Kubiatowicz, David Chaiken, and Anant Agarwal. The Alewife CMMU: Addressing the Multiprocessor Communications Gap. In *HOTCHIPS*, August 1994.

[33] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. In *Proceedings International Symposium on Shared Memory Multiprocessing*, Japan, April 1991. IPS Press.

[34] J. Kuskin *et al.* The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, Chicago, IL, April 1994. IEEE.

[35] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[36] Butler Lampson, William Weihl, and Eric Brewer. 6.826 Principles of Computer Systems. Research Seminar Series MIT/LCS/RSS 19, Massachusetts Institute of Technology, July 1992.

[37] Charles E. Leiserson *et al.* The Network Architecture of the Connection Machine CM-5. In *The Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1992.

[38] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[39] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[40] S. Lo and V. Gligor. A Comparative Analysis of Multiprocessor Scheduling Algorithms. In *7th International Conference on Distributed Computing Systems*, pages 356–363. IEEE, Sept. 1987.

[41] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive Algorithms for On-line Problems. In *Proceedings of the 20th Annual Symposium on Theory of Computing*, pages 322–333, Chicago, IL, May 1988. ACM.

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

[43] 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.

[44] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, Jul 1991.

[45] Bodhisattwa Mukherjee and Karsten Schwan. Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 59–66, July 1993. Also available as Technical Report GIT-CC-93/17, Georgia Institute of Technology.

[46] John Nguyen. *Compiler Analysis to Implement Point-To-Point Synchronization in Parallel Programs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, August 1993.

[47] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[48] 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 (ISCA)*, Chicago, IL, April 1994. IEEE.

[49] Michael L. Scott and John M. Mellor-Crummey. Fast, Contention-Free Combining Tree Barriers. Technical Report TR-429, University of Rochester, June 1992.

[50] Z. Segall and L. Rudolph. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *11th Annual International Symposium on Computer Architecture*, pages 340–347. IEEE, June 1984.

[51] Dennis Shasha and Nathan Goodman. Concurrent Search Structure Algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, March 1988.

[52] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[53] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *Society of Photocoptical Instrumentation Engineers*, 298:241–248, 1981.

[54] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, May 1992.

[55] Jae-Heon Yang and James H. Anderson. Fast, Scalable Synchronization with Minimal Hardware Support. In *12th Symposium on Principles of Distributed Computing (PODC)*, Ithaca, NY, 1993. ACM.

[56] Donald Yeung and Anant Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, May 1993. ACM.

[57] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[58] J. Zahorjan and E. Lazowska. Spinning Versus Blocking in Parallel Systems with Uncertainty. Technical Report TR-88-03-01, Dept. of Computer Science, University of Washington, Seattle, WA, Mar 1988.