OPTIMIZED MULTIPROCESSOR COMMUNICATION AND SYNCHRONIZATION USING A PROGRAMMABLE PROTOCOL ENGINE

John Heinlein

Technical Report No.: CSL-TR-98-759

March 1998

This dissertation was supported by an Air Force Laboratory Graduate Fellowship (administered by the Air Force Office of Scientific Research), by an Intel Foundation Fellowship, and by the Defense Advanced Research Projects Agency, which funded the FLASH project under DARPA Contract DABT63-94-C-0054.

Optimized Multiprocessor Communication and Synchronization Using a Programmable Protocol Engine

John Heinlein

CSL-TR-98-759

March 1998

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Gates Computer Science Building, #408
Stanford University
Stanford, CA 94305-9040
pubs@shasta.stanford.edu

Abstract

In recent years, multiprocessor designs have converged towards a unified hardware architecture despite supporting different communication abstractions. The implementation of these communication abstractions and the associated protocols in hardware is complex, inflexible, and error prone. For these reasons, some recent designs have employed a programmable controller to manage system communication. One particular focus of these designs is implementing cache coherence protocols in software. This dissertation argues that a programmable communication controller that provides cache coherence can also effectively support block transfer and synchronization protocols. This research is part of the FLASH project, a major focus of which is exploring the integration of multiple communication protocols in a single multiprocessor architecture.

In our analysis, we examine the needs of protocols other than cache coherence to identify the requirements they share. The interface between the processor and controller is one critical issue in these protocols, so we propose techniques to export such protocols reliably, at low overhead, and without system calls. Unlike most prior studies, our approach supports a modern operating system with features like multiprogramming, protection, and virtual memory.

Our study focuses in detail on two classes of communication that are important for large scale multiprocessors: block transfer and synchronization using locks and barriers. In particular, we attempt to improve the performance of these classes of communication as compared to implementations using only software on top of shared memory. For each protocol we identify the critical metrics of performance, explore the limitations of existing techniques, then present our implementation, which is tailored to leverage the programmable communication controller. We evaluate each protocol in isolation, in the context of microbenchmarks, and within a variety of applications.

We find that embedding advanced communication and synchronization features in a programmable controller has a number of advantages. For example, the block transfer protocol improves transfer performance in some cases, enables the processor to perform other work in parallel, and reduces processor cache pollution caused by the transfer. The synchronization protocols reduce overhead and eliminate bottlenecks associated with synchronization primitives implemented using software on top of shared memory. Simulations of scientific applications running on FLASH show that, in many cases, synchronization support improves performance and increases the range of machine sizes over which the applications scale. Our study shows that embedded programmability is a convenient approach for supporting block transfer and synchronization, and that the FLASH system design effectively supports this approach.

Keywords & Phrases: multiprocessors, FLASH, cache-coherent shared memory, message passing, synchronization

Acknowledgements

I have had the privilege to carry out my graduate studies among some of the most talented researchers in the computer architecture field. I especially thank my advisor, Anoop Gupta, whose guidance has helped me grow as both a researcher and a person. His advice and feedback on this dissertation itself has significantly improved its quality. I am also grateful to John Hennessy, my secondary advisor. Despite the pressures of being computer science department chair and then engineering dean, John was always available to give me his invaluable guidance and support. I thank Bruce Wooley for chairing my orals committee and serving on my reading committee, which were very generous given his equally taxing responsibilities as acting electrical engineering department chair.

Besides those faculty on my reading committee, I would like to acknowledge the contributions of two others as well. First, Mark Horowitz, who served as the day-to-day leader of the FLASH project. Mark's seemingly limitless knowledge and experience, as well as his skill at managing the project made a difficult task achievable and fun. I also benefitted greatly from the advice, friendship, and collaboration of Mendel Rosenblum, who brought a valued alternative perspective to the project.

I feel particularly lucky to have received the guidance of DASH veteran Kourosh Gharachorloo, who worked closely with me at Stanford throughout the work on block transfer and later during my internship at Digital Equipment Corporation's Western Research Laboratory (WRL). His attention to detail has enriched my graduate training. His insight is reflected in particular in the techniques presented in Chapter 4.

I was also fortunate enough to have a wonderful group of fellow researchers with whom I was able to work. First and foremost, the FLASH team was responsible for developing the novel architecture studied in this research: Joel Baxter, Jules Bergmann, Mark Heinrich, Hema Kapadia, Jeffrey Kuskin, David Ofelt, David Nakahira, and Richard Simoni. I also acknowledge two other colleagues, Steven Woo and Chris Holt, who graciously helped me unravel the SPLASH applications I studied. I am also grateful for having been able to work closely with a "second family" as well, Mendel Rosenblum's Hive and SimOS groups: Robert Bosch, Edouard Bugnion, John Chapin, Scott Devine, Kinshuk Govil, Steve Herrod, Beth Seamans, Dan Teodosiu, Ben Verghese, Ben Werther, and Emmett Witchel. John Chapin collaborated on the operating system and interface issues for the protocols we study. Robert Bosch contributed to the performance evaluation of memory copy.

Of all these, I would like to specially acknowledge my two sets of officemates: from the CSL Trailer, David Ofelt and Jeffrey Kuskin; and from Gates 354, Robert Bosch and Steve Herrod. I learned a great deal from them especially, and I sincerely valued their friendship and support over these many years. Special thanks also go to Joel Baxter and Mark Heinrich whose endless simulator work and unfailing assistance were invaluable in supporting this research.

I thank Charlie Orgish and Thoi Nguyen for helping me sleep at night knowing my data was safe, but even more importantly for being wonderful friends. Throughout my studies and my work as CSL student bureaucrat, I benefited from the care and help of a number of administrators: Lori Balough, Darlene Hadding, Margaret Rowland, Naomi Schulman, Eileen Schwappach, Terry West, and many others. From out of the blue I was blessed with the advice and kindness of Nick McKeown and Murray Warren, who helped me along my journey in many ways.

I gratefully acknowledge support for my graduate studies from an Air Force Laboratory Graduate Fellowship (administered by the Air Force Office of Scientific Research), from an Intel Foundation Fellowship, and from the Defense Advanced Research Projects Agency, which funded the FLASH project under DARPA Contract DABT63-94-C-0054. The completion of my dissertation took place at Transmeta Corporation. I cannot adequately express my gratitude for the support and understanding I received from my supervisors Craig Anderson and Jack Guskin, especially as the process dragged on beyond all expectation.

I thank Michelle Stohlmeyer for her loving and patient support during the often difficult process of finishing my dissertation. Her empathy (not to mention her excellent care packages) kept me going in the final stages.

Finally, I am deeply grateful for the unfailing love and support of my parents, John and Janet, and my brother Jason. The nurturing environment they provided is undoubtedly responsible for my accomplishments in life, and my love for them cannot be overstated.

To my parents John and Janet
who gave me every opportunity in the world.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A critical problem in multiprocessors is managing communication between the processors in the system. Individual processor performance is important, but it alone is not sufficient to achieve scalable system performance. To address the communication problem, multiprocessor systems research has explored a broad range of design styles and, in turn, communication abstractions presented to the application. Among these models are shared memory, message passing, dataflow, systolic arrays, SIMD, and many others, each of which has advantages for certain classes of programs [Fly66, Vee86, AAG$^+$87, AS88, Bla90, Int91, Len92, HT93].

Over time, two models have emerged as the predominant ones supported by multiprocessor systems: cache-coherent shared memory and message passing. Shared memory provides the programmer with a simple memory abstraction similar to a uniprocessor that is particularly well suited for programs that exhibit dynamic communication behavior or fine grain sharing. Certain other types of communication such as the transfer of coarse grain data can sometimes be achieved more efficiently through message passing, though it significantly increases the data management burden on the application programmer.

The complementary nature of the shared memory and message passing communication styles, in conjunction with the convergence of underlying hardware mechanisms used to implement each model has led to a surge of interest in hybrid architectures that support both styles efficiently. Furthermore, the complexity of implementing the protocols in hardware has motivated the development of hybrid hardware/software solutions to manage communication [ACD$^+$91, Cra93, ACD$^+$95].

As one recent approach, several systems have been designed using a programmable controller to manage system communication [ACD$^+$91, ACD$^+$95, KOH$^+$94]. One particular focus of these systems is implementing cache coherence protocols in software. This study argues that a programmable communication controller that provides cache coherence can also effectively support block transfer and synchronization protocols. Our focus is on one such system in particular, the FLASH Multiprocessor, which includes a programmable protocol engine to serve as the memory system and network interface controller. FLASH (*FL*exible *A*rchitecture for *SH*ared memory), was designed by a team of researchers at Stanford University between 1992–1997. This dissertation complements previous studies of FLASH, which focus on its architecture and its use with a range of cache coherence protocols [KOH+94, HKO+94, MOH96, Kus97, Hei].

Since the introduction of FLASH, other commercial and research systems have been proposed that incorporate some of the same design characteristics, including the use of a flexible protocol engine. Among these systems are Typhoon [RLW94, RPW96] and the Sequent NUMA-Q [LC96]. As in FLASH, the published research for these systems has also focused primarily on shared memory, though many of the issues we address also apply to those systems. Thus, our study of "alternate" protocols may be applicable in a wider context than just the FLASH project.

1.1 Using a Programmable Protocol Engine

Traditionally, within the core of a multiprocessor system is a hardware unit designed to implement the communication model provided by the machine. For example, in the case of cache-coherent shared memory, the communication controller transparently manipulates data to support the illusion of shared memory. For message passing, it accepts explicit requests for data transfer and moves data asynchronously.

Instead of the traditional hardware-only approach, the system we study uses a communication controller containing a programmable engine at its core. As we describe below, this approach provides a number of benefits over the traditional design. We briefly explore the characteristics of these systems in general, then introduce the particular design of the FLASH system.

1.1.1 Why Embed Programmability?

Embedding programmability to replace a pure hardware solution is not a new idea. The use of microcode to manage the internal control signals of a processor was proposed in 1949 [HP90]. The approach we study, however, uses embedded programmability in a new context: within the communication controller in a multiprocessor system. Flexibility in this context seems a natural extension of computer system design trends occurring in recent years. Several examples of these trends are as follows:

The Alewife system designed at MIT allows software control over communication protocols by supporting fast processor interrupts. Alewife's approach thus *shares* a single processor to perform both computation and protocol processing tasks [ACD+91, ACD+95].

Multiprocessor bus controllers continue to grow in complexity and functionality, especially those for high-end servers. Current designs include support for features such as high performance I/O management, RAS (Reliability, Availability, and Serviceability), and seamless multiprocessing expandability. These features have reached a level of complexity where an implementation based partially or entirely on software may soon be a reasonable design trade-off.

The Stanford DASH and the Sun S3.mp systems both integrate a controller with some degree of programmability within their coherence protocol units. DASH uses a hybrid approach in which most functions are provided in hardware but key protocol decisions are stored in a table. This allows only limited customizations, such as enabling some protocol errors to be fixed [Len92]. S3.mp provides a slightly more powerful model closer to a traditional microcode approach [NAB$^+$95].

Flexibility embedded in the system enables a broad range of advantages over a purely hardware approach. First and foremost it allows one architecture to support a range of different communication models. Hardware approaches could support multiple models, but they must be selected and fixed in advance. On the other hand, a system with embedded programmability allows the communication protocols to be changed even after the system has been designed and built.

Late customization is valuable for a variety of reasons, among them: *(i)* to fix bugs in the protocol, *(ii)* to enable low-level performance optimizations, and *(iii)* to allow communication extensions for particular applications or even entirely new protocols. In addition, protocol development in software is convenient since one can use traditional software development and debugging techniques. The challenge in a system with embedded flexibility is achieving high performance, since an implementation incorporating software may be less efficient than one based purely on hardware.

1.1.2 The FLASH Approach

FLASH was designed from the beginning to balance the goal of integrating flexibility with the desire for high performance. It is based on a programmable communication controller, MAGIC (Memory And General Interconnect Controller), placed centrally in the node and given high-performance access to memory, network, I/O, and the main processor. One of the key innovations in MAGIC is its support for the parallel handling of control processing and data movement. Its centerpiece is a programmable Protocol Processor that performs the control processing in software. Alongside the Protocol Processor, MAGIC contains optimized hardware units to swiftly move data between its communication interfaces. This hybrid design is enables protocols with a wide range of characteristics to achieve high performance.

1.2 Dissertation Focus

The focus of this dissertation is to study the advantages and design issues of using a programmable controller to support protocols other than cache coherence. We argue that through careful design of the controller and protocols the same hardware designed for cache coherence can support other classes of communication as well. We support our thesis by analyzing alternate protocols from several different fronts, which we introduce below:

1.2.1 Processor/Controller Interface Issues

While the introduction of a programmable controller opens up a range of possibilities of communication functionality and performance, it also brings with it many new problems. For communication support in the memory system to be useful, the processor must be able to access it at low overhead. To accomplish this goal, we describe techniques that export these protocols at user level, i.e., without system calls. Avoiding system call overhead increases the range of protocols that show benefits from implementation on the communication controller.

However, unlike past studies that have considered systems running in single-user mode or in a restricted system environment, our goal is to provide support for a modern operating system. To bridge this gap, our research proposes techniques to support operating system features like multiprogramming, protection, and virtual memory.

1.2.2 Requirements of Alternate Protocols

One goal of embedding programmability in the communication controller is to support a range of protocols effectively. Nonetheless, the design of these systems tends to focus more strongly on the requirements of one protocol in particular, in this case cache coherence. One focus of this study is to understand in a similar way the controller features which are particularly valuable for alternate protocols.

This analysis studies the FLASH system and its node controller, MAGIC, focusing on the features that are particularly important for alternate protocols. We analyze a range of specific alternate protocols that one might support with a programmable controller, identifying their requirements and their amenability to implementation on FLASH. In cases where the FLASH system design limits the full generality of these protocols we discuss the tradeoffs associated with extending the controller and propose alternatives that can address the limitations without controller extensions.

1.2.3 Detailed Protocol Studies

Finally, our study focuses in detail on two particular classes of communication which are important for large scale multiprocessors. We find that each of these classes encounters disadvantages when implemented on top of shared memory, in part because their communication does not exactly match that provided by the cache coherence protocol.

First we present the mechanisms used to support efficient block transfer in FLASH. The protocol we implement and study provides *memory copy* functionality, which can be used to support message passing as well as accelerate block transfer communication in shared memory programs. Our protocol delegates the data transfer functionality to MAGIC which implements it with protocol software. In addition to describing the transfer implementation itself, we also explain how the protocol integrates block transfer with cache coherence, which is critical to enable its use within shared memory applications.

In our experiments we compare the performance of the FLASH memory copy primitive with processor-based implementations, including those using prefetching. Our results suggest that implementing block transfer support in MAGIC improves transfer performance in many cases, enables the processor to perform other work in parallel, and reduces processor cache pollution caused by the transfer.

We also study the design of two synchronization primitives, *locks* and *barriers*. These primitives are typically implemented on top of shared memory and as a result often perform poorly due to artifactual communication associated with cache coherence. Our design targets these artifacts; by eliminating them we improve synchronization performance and characteristics.

Synchronization protocols highlight the benefits of the careful implementation of embedded programmability in MAGIC. These protocols are very lightweight, with very different characteristics than the block transfer protocol. Nonetheless, the flexibility of FLASH enables high performance for these protocols as well. Our results show that synchronization support from the programmable controller improves performance of some applications from the SPLASH-2 benchmark suite [WOT$^+$95], especially at larger machine sizes.

1.3 Contributions

The primary contributions of this dissertation are the following:

- We propose mechanisms to allow the processor and controller to cooperate effectively. One important facet of this cooperation is permitting the processor to request communication features at low overhead. Unlike most prior studies, our techniques provide this capability while maintaining the integrity of processor operating system features such as protection, virtual memory, and multiprogramming.

- We examine the implementation details of the communication controller that impact the performance or functionality of other protocols differently than they impact cache coherence. To this end, we consider a range of communication protocols to determine the requirements they share, and discuss the impact on those protocols of controller design features or restrictions.

- We study the design trade-offs for a custom FLASH memory copy protocol, and present an implementation that leverages the support provided by the programmable communication controller. A major focus of this design is on efficiently integrating the protocol with cache coherence to enable the primitive to be used in a wide range of situations. This goal raises major challenges, especially in achieving the integration while maintaining high performance.

- We present custom lock and barrier synchronization primitives for FLASH. These primitives are targeted to improve the performance of synchronization as compared to shared memory implementations by eliminating artifactual communication caused by cache coherence. We examine several conventional approaches to identify their benefits and limitations. Then we describe the implementation of the custom protocols in detail, identifying how the primitive improves performance by matching the inherent communication the operation requires.

1.4 Organization

This dissertation is organized as follows:

Chapter 2 begins by describing the motivation for incorporating a programmable communication controller inside a multiprocessor. Then it describes the solution we consider, the FLASH Multiprocessor, focusing closely on the characteristics that are relevant to the alternate protocols we study.

Chapter 3 describes the design space and motivation for alternate uses of the flexible communication controller beyond cache coherence. This chapter focuses on the protocols that we implement and study in detail in later chapters: block transfer/message passing, and lock and barrier synchronization.

In Chapter 4 we explore issues arising from the division of functionality between the main processor and communication controller. In particular we present techniques that enable efficient communication between the processor and MAGIC without system calls. We also describe a range of approaches to permit MAGIC to interact with processor features such as multiprogramming, virtual memory, and protection.

Then we present the two protocol classes in detail. Chapter 5 presents the FLASH memory copy protocol, which can be used to provide message passing functionality as well as accelerate block transfer within shared memory applications. Chapter 6 presents FLASH lock and barrier synchronization protocols, designed to improve synchronization performance and characteristics beyond what can be achieved through traditional primitives based on shared memory. In each case, we first describe the design and goals of the protocol in detail, then present its implementation. Finally, we evaluate each protocol in isolation, in the context of microbenchmarks, and within a variety of applications.

Chapter 7 describes other protocols that are amenable to using a programmable controller such as MAGIC. We describe issues associated with “active messages”, which pose some unique implementation challenges, as well as other protocols that may be promising future research directions.

Finally, Chapter 8 summarizes the conclusions of the dissertation.

Chapter 2

The FLASH Multiprocessor

This chapter describes the Stanford FLASH Multiprocessor, a high-performance, scalable parallel computer. This dissertation uses the FLASH system as a context for the study of communication and synchronization protocols. We begin by briefly describing the background behind the FLASH system to motivate its fairly unique design. Next the FLASH architecture is described, with emphasis on the features and characteristics of particular interest for supporting advanced communication and synchronization primitives. A very complete presentation of the FLASH architecture and prototype appears in [Kus97].

2.1 Background and Motivation

In designing a multiprocessor, we are presented with a wide design spectrum from which to choose. We focus on two key issues: the ability of the system to scale and the different choices of communication model to present to the user. We also present the design point offered by FLASH's direct predecessor, the DASH system, before finally describing the FLASH system itself.

In small scale multiprocessors, the processors typically share a single bus. This style, called a bus-based or "symmetric" multiprocessor (SMP) is illustrated in Figure 2.1. SMPs can be fairly simple to design, and for small machine sizes are very effective. However, the single bus fundamentally limits their scalability. Beyond a certain size the load offered by additional processors overwhelms the bus and effective request latency increases. The exact size when this occurs depends on the performance of the processors and bus, though usually it less than 16–32 processors. Systems of this design are sometimes known as Uniform Memory Access (UMA) systems since all memory accesses take (essentially) the same latency to be satisfied.

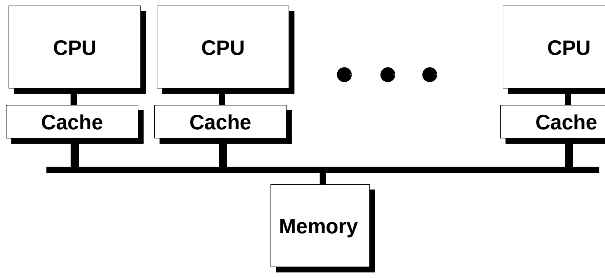


Figure 2.1: Symmetric multiprocessor architecture.

Recently, the trend in the design of multiprocessors is towards scalable architectures. To achieve scalability, SMPs have gradually been replaced by systems with several important modifications. First, the system is split into groups of resources, called *nodes*. Each node contains a small number of processors, typically between 1–4. Instead of the single memory system in an SMP, memory is distributed around the system, with each node containing a portion of the overall machine memory. Distributing the memory increases aggregate bandwidth and allows the system memory to scale without requiring expensive memory architectures such as those used in many supercomputers. Second, the single bus is replaced by a more scalable interconnect between the nodes, one that can offer more bandwidth as the system scales. With the scalable interconnect, a communication controller is introduced to manage the network. Figure 2.2 illustrates this architecture.

These scalable systems provide applications with one of the two predominant communication paradigms: *shared memory* or *message passing*. In a shared memory system, the communication controller "hides" the distribution of the machine's physical memory by transparently communicating with the correct node (and thus memory module) for processor requests to remote memory. Shared memory provided on a distributed memory system is often called Distributed Shared Memory (DSM), or Non-Uniform Memory Access (NUMA). The latter name arises because, unlike SMPs, the time to access memory depends on the location of the memory being accessed.

In some shared memory systems nodes are also permitted to hold data from remote memory locations in their cache. This can improve performance by eliminating repeated expensive misses for remote data. Caching remote memory introduces the problem, however, that modifications made to shared memory may not be globally visible. If this occurs, nodes around the system may see different values for the same memory location. This problem, known as *cache coherence*, requires that either the processor or the system hardware take special steps to assure that when a line is modified the currently cached copies are eliminated. Systems providing cache coherence support are often called Cache Coherent Non-Uniform Memory Access (CC-NUMA) machines.

The second communication paradigm found in scalable systems is *message passing*. In a message passing system, the distribution of the memory is made visible to the user; only local memory can directly be accessed by the processor using load and store instructions. Communication between

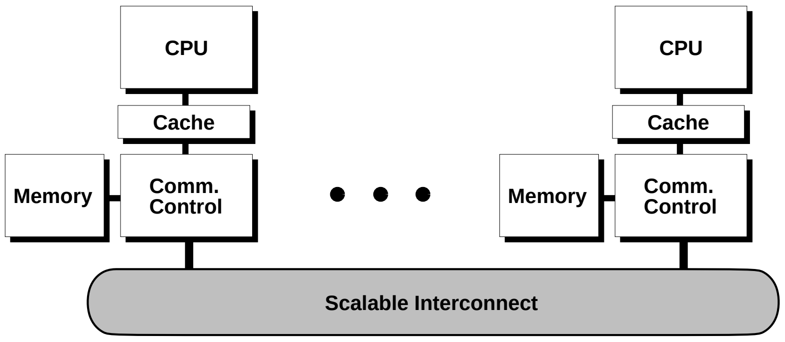


Figure 2.2: Scalable multiprocessor architecture.

nodes is accomplished through explicit *messages*. Message passing offers the system or application the ability to customize the communication granularity and timing; the drawback is that it increases the burden on the application to manage the communication. In Section 3.1 we compare these two models in further detail.

From a system design perspective, what is interesting about these two paradigms is that despite the vast differences in the interfaces they export to the user, they both utilize an architecture similar to the one in Figure 2.2. The main difference is in the design and operation of the communication controller, which usually supports one model or the other. At the beginning of the FLASH project, systems being designed elsewhere highlighted the differences between these two design styles and provided mechanisms for one or the other, instead of focusing on the extensive commonality between them [DCF$^+$89, Int91, Thi91, Bec92, Cra93]. One of the main goals of the FLASH project is to consider the design of scalable multiprocessors, in particular the communication controller and the protocols they use, to see if both of these models (and potentially others) can be efficiently supported by a *single* system.

2.1.1 The DASH Multiprocessor

To explore briefly how we might design the FLASH system, we first consider the architecture of its predecessor. In 1989, researchers at Stanford University began the design of the DASH multiprocessor (*D*irectory *A*rchitecture for *SH*ared memory) [LLG$^+$90, LLG$^+$92, Len92]. DASH aimed to demonstrate a real implementation of a scalable hardware-supported cache-coherent shared memory system.

DASH is built by interconnecting small-scale bus-based commercial multiprocessors; it is comprised of SGI 4D/240 systems, which contain 4 processors each. To these systems, called a *cluster*, the DASH designers add two boards to support cache coherent distributed shared memory [Len92]. DASH uses a directory-based approach to cache coherence, which was described as early as 1978 by Censier and Fautrier [CF78]. In that approach, each node maintains a table called a *directory* that tracks where lines are cached in the machine. In DASH, the directory is a dedicated memory array (implemented in SRAM), which stores the list of sharers in a bit vector.

DASH shows the benefits of hardware cache coherence, and that its complexities are manageable. The specific implementation of DASH has two main limitations, however. First, the system is only designed to scale to 64 nodes. In particular, the bit vector directory format used in DASH is inappropriate for larger size machines. Second, this protocol is implemented through on-board tables indicating the actions the hardware should take based on network and processor events. These tables are designed to encapsulate the hardware actions needed by the DASH protocol, but provide only limited flexibility if protocol modification should be desired for research purposes or needed to fix unexpected bugs.[1] The ability to modify the protocol after the fact is a serious concern given the difficulty in adequately simulating and verifying large parallel machines.

2.1.2 The FLASH Approach

Given this background as context, the FLASH project focuses on two main goals. The first is to study the design of scalable multiprocessor systems, with emphasis on the communication controller, to see if a single system can effectively support both cache-coherent shared memory and message passing. The second is to implement cache coherence protocols in a more flexible way, so the protocol may later be corrected, optimized, or replaced altogether.

FLASH addresses these goals by replacing the hard-coded table-based coherence protocol of systems like DASH with an embedded processor capable of handling processor and network events in *software*. Along with this processor, FLASH provides specialized hardware to improve protocol processing throughput. This style of design allows the coherence protocol to be modified and extended as needed. In addition, it permits the implementation of so-called *alternate* protocols such as message passing, scalable synchronization, fault tolerance and recovery, and performance monitoring.

2.2 FLASH Overview

The FLASH Multiprocessor is a high performance parallel computer system being designed and implemented by a team of researchers at Stanford University [KOH$^+$94, Kus97]. FLASH (*FL*exible *A*rchitecture for *SH*ared memory) consists of a scalable array of processing nodes connected by a low-latency, high-bandwidth communication network. Each node in the multiprocessor contains all the major components of a modern high-performance scientific workstation, with the addition of a custom, programmable node controller to provide communication and other functions. The FLASH architecture is illustrated in Figure 2.3.

[1]Though DASH does not directly provide the ability to support models other than shared memory, it does provide some optimizations in the protocol for data movement and locks.

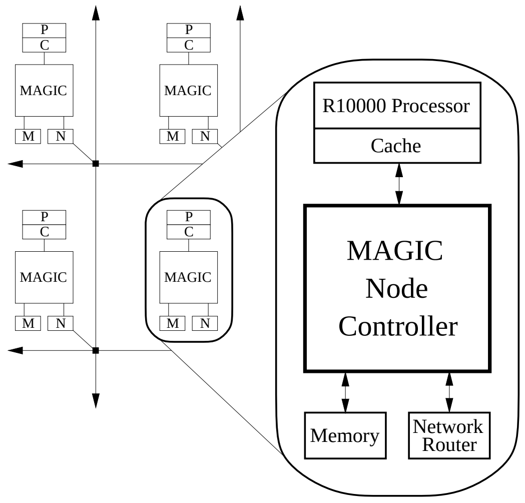


Figure 2.3: FLASH Node Architecture.

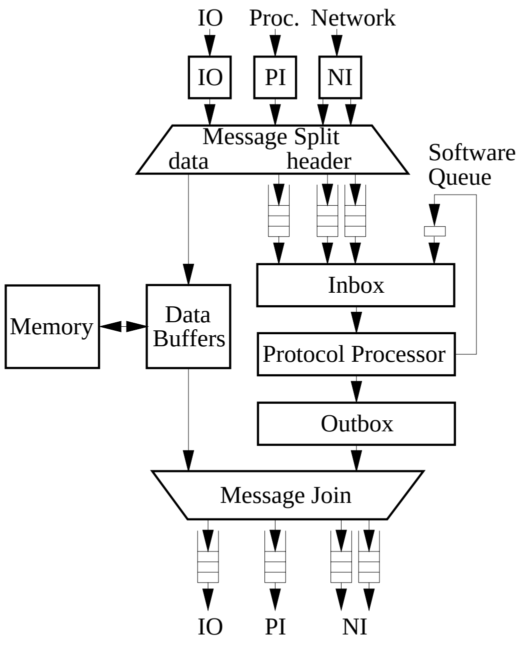


Figure 2.4: MAGIC Microarchitecture.

2.2.1 FLASH System Architecture

Each FLASH node contains a single processor, the MIPS R10000.[2] The R10000 is a dynamically scheduled superscalar processor that provides aggressive performance for both integer and floating point code, achieving a SPECint95 of 8.85 and SPECfp95 of 13.8 (17.5 predicted) [MIP96]. This enables the FLASH system to efficiently run not only scientific applications, which generally have heavy floating point demands, but integer-based applications as well. The R10K provides on-chip instruction and data caches, each 32 KB, as well as a variable-sized, processor-managed secondary cache. In the initial FLASH prototype, the secondary cache of the R10K is 1 MB.

The FLASH node also contains a large amount of DRAM, similar to workstations. In FLASH, this memory is part of the machine-wide distributed main memory as described earlier. Logically, the memory on a node is one piece of a contiguous physical address space beginning with address zero on node zero and ending with the physical memory on the highest numbered node. The FLASH system reserves a small portion of the memory on each node for protocol code and storage. FLASH uses this protocol storage in part to implement a directory for maintaining cache coherence, similar to DASH. By using main memory instead of a specialized memory, protocol storage can grow or shrink as needed to support arbitrary communication and synchronization protocols.

[2]The use of a uniprocessor node in FLASH is not a fundamental restriction, but was a design decision partly driven by implementation practicality concerns. The FLASH architecture can support a multiprocessor node with only localized modifications.

In conventional workstations, dedicated hardware chipsets are typically used to control memory, I/O devices, and other board-level resources. The key innovation in the FLASH system is the use of a custom node controller, named MAGIC (*M*emory *A*nd *G*eneral *I*nterconnect *C*ontroller). MAGIC comprises a programmable processor optimized for executing protocol operations and dedicated data paths which provide low latency communication between the main processor, memory, and the communication ports of the node. Figure 2.4 shows the microarchitecture of MAGIC.

Besides the interface to the processor, MAGIC provides two IO interfaces on a FLASH node. For high-bandwidth, low-latency communication between nodes in the system, FLASH uses the SGI CrayLink network technology (formerly nicknamed “Spider”) [Gal96]. CrayLink is configured as a “fat hierarchical hypercube topology” with data transfer bandwidth of 800 MB/s per endpoint. Figure 2.4 provides a logical view of the queues in MAGIC that are used to buffer incoming and outgoing protocol messages. We exploit the virtual lane capability of the network, which provides four incoming and outgoing network queues. This simplifies the solution for deadlock in most protocols since separate queues can be used for request and reply messages [LLG$^+$90].

The second network is a PCI interface for use in comparably low-bandwidth I/O devices such as disk, console, graphics, Ethernet, etc. Section 2.3 describes the architecture of MAGIC in more detail, to provide background for the detailed protocol descriptions and tradeoffs described later in this thesis.

2.2.2 Communication Protocol Terminology and Semantics

In this dissertation, we refer to a *protocol* to mean a specification describing the interaction among nodes in the FLASH system to accomplish a particular communication task. For example, a cache coherence protocol describes interactions between nodes aimed at providing a consistent view of memory despite caching on remote nodes. Specifically, the protocol encompasses the behavior which the participants provide in response to messages, as well as the kinds of messages they can send. In practice, protocols can be implemented in a variety of ways: using hardware, software, or a combination of both.

In the FLASH system, protocols are tightly integrated into the node through the hardware and software communication features provided by the MAGIC chip. Internally, MAGIC expresses the requests from its different interfaces in a common format, called *messages*. A message is the smallest unit of communication on FLASH, consisting of two parts: The *header* portion describes the contents of the message, its sender, receiver, data length, and other attributes useful to MAGIC. The *data* portion contains the message payload, which in FLASH is either empty (no payload), one doubleword (eight bytes), or one cache line (128 bytes). Note that these messages are not the same as application-level messages exchanged in a message passing communication model on the main processor. Rather, the messages described here are short communications used internally by the

FLASH system to implement more complicated communication operations requested by the main processor.[3]

Similar to the Active Message model [vECGS92], MAGIC invokes a short segment of code called a *handler* to process each message it receives. Unlike the original description of active messages which sends the actual program counter in the header of the request [vECGS92], FLASH provides an encoded *message type* instead. This message type is used along with two other attributes of the message (whether it comes from the processor or network, and whether the address is local or remote) to select a group of four possibly matching handlers from a dispatch table. These candidate handlers are then considered more closely using a number of additional attributes: a protection check (whether the sender is within the same fault containment boundary as the receiver), the availability of data buffers, additional parts of the address, and several others. By providing handler dispatch functions in hardware, a number of different attributes can be tested in parallel rather than using software-based dispatch, which would be significantly slower.

In the FLASH system, the implementation of a protocol consists of three parts. First, a set of unique message types is selected, representing the kinds of requests a node can make. When multiple kinds of communication coexist on the machine (a major focus of later chapters), each takes a subset of the message type name space for its use. Second, a collection of software handlers is created to satisfy these different requests. Finally, the dispatch table is created that describes the correspondence between the handlers and the situations in which they should be invoked (which includes the message type and the other factors described above).

Protocol handlers may need to carry out a wide range of actions to satisfy the request depending on the particular services needed. Handlers may send messages to other nodes, read and write protocol state affected by the request, perform memory operations, make requests of the local processor, or carry out other tasks internal to MAGIC. In many cases, the original request generates a response to the sender, eventually causing a handler to execute on the protocol processor of the sender's MAGIC chip. In Section 2.3.6, we consider some restrictions that must be placed upon the handlers in the FLASH system to avoid deadlock.

2.2.3 Cache Coherence

In this section, we describe the cache coherence protocol in FLASH. This forms the basis on top of which the cache-coherent block transfer and other protocols are built in the remainder of this dissertation. The coherence protocol consists of two main parts. First we consider the format of the directory which is used to track outstanding copies of lines in the system. Then we consider the algorithm through which the system maintains coherence, based on the information in the directory.

[3]Later, in discussing message passing protocols in FLASH we generally refer to these application-level messages as *block transfer*, so the distinction should be unambiguous.

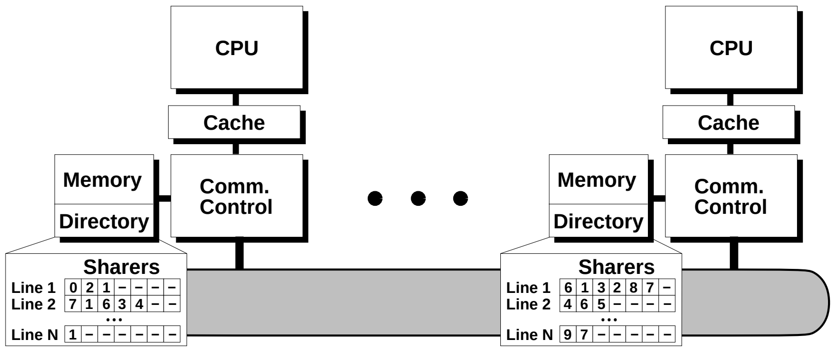


Figure 2.5: Conceptual illustration of a cache coherence directory.

Directory Format

The FLASH system is fundamentally designed to allow many different protocols and directory formats to be used. Figure 2.5 illustrates a generic directory structure, showing how each node maintains directory state for its share of the machine's memory. As part of the project, other researchers have developed several completely different cache coherence protocols for FLASH to study the trade-offs in detail. For this dissertation, we have chosen to use one of the more scalable of these directory formats, *dynamic pointer allocation*, designed to support machines up to several hundreds or even thousands of nodes. The FLASH implementation of this protocol is the work of Mark Heinrich.

Dynamic pointer allocation was originally developed as part of the DASH project by Richard Simoni as a way to surpass the limited scalability of bit-vector based approaches [Sim92]. This approach takes advantage of the sharing characteristics of typical applications: most memory lines are only shared by one or two nodes at any time, while relatively few lines are shared more widely [WG89]. Unlike bit-vector or limited-pointer based formats [ASHH88], which allow for a conservative amount of caching of *each* memory line, dynamic pointer allocation uses a minimal per-line directory entry. Then it adds a pool of pointers[4] shared between all the lines and allocated on demand (i.e., dynamically), for use in handling lines with more widespread sharing.

The FLASH implementation of dynamic pointer allocation maintains a directory entry per memory line capable of tracking the line being cached by the local processor and one remote processor. It also maintains an array of bits used to track the state of the line (e.g., whether it is modified, whether it is busy with coherence actions, etc.). Finally, it also holds a total sharer count and the beginning of a linked list of pointer entries, which are allocated as needed from the large shared pool.

[4]In this context, a *pointer* refers to an indication of a single remote sharer of a line.

Coherence Algorithm

The FLASH coherence protocol is a (MESI) exclusive ownership-based protocol. Lines which are read-only may be widely shared in the system, but before a line may be modified, all outstanding copies are eliminated so only a single writable copy exists. When ownership of a line is requested, FLASH eliminates outstanding copies by sending *invalidations* to remote caches currently holding the data, instructing the cache to discard the line. To improve performance, relaxed consistency models such as release consistency [Gha95] allow optimizations such as permitting writes to occur in parallel with sending the invalidations; FLASH can support many of these optimizations. As alluded to in Section 2.3.3, restrictions imposed by the processor prevent us from using an update-based approach in which modifications to the line are sent to current holders to keep their copy current.

One of the difficulties that arises in a cache coherence protocol is managing the asynchrony that occurs from many processors simultaneously sending requests. If multiple references occur to the same line, they may conflict. To make this problem more manageable, the protocol marks a line as *pending* in the directory when coherence actions are in progress (such as sending invalidations to a list of sharers in preparation for providing exclusive access). Requests for lines which are marked as busy are refused with a negative acknowledgement (NAK) and are forced to retry.

2.3 MAGIC (Memory and General Interconnect Controller)

This section describes the microarchitecture of MAGIC in more detail, to provide a basis for the description of the protocols in later chapters. We focus on the portions which most directly impact the protocols we implement and study; this is not intended to be a complete description of MAGIC. More details can be found in the publications on FLASH [KOH$^+$94, HKO$^+$94, HGDG94, Kus97, Hei]. First we provide an overview of the MAGIC microarchitecture and illustrate the way control and data are handled differently. Next we focus on the custom protocol processor (PP) inside MAGIC, with emphasis on its specialized support for communication protocols. We describe the processor interface (PI) in some detail, since it impacts the kinds of operations our protocols can use to request data from and supply data to the processor. Finally, we explain the data buffer (DB) logic which is used to stage data as it passes through MAGIC, highlighting certain special features provided by the DB for efficient protocol data movement.

2.3.1 MAGIC Microarchitecture Overview

A major goal in designing MAGIC was to optimize both the protocol processing of incoming messages and the transfer of data between its interfaces. For example, MAGIC may need to decide the correct actions to maintain coherence when a request arrives, and may need to move a cache line of data as part of the request. To accomplish this goal, MAGIC splits an incoming message into its control and data components, as illustrated in Figure 2.4. Data processing is provided by dedicated data paths that transfer data between interfaces in a pipelined fashion. Control processing is accomplished in parallel by the programmable protocol processor. The protocol processor controls macro-operations on the data paths using specialized instructions. These operations can cause, for example, an entire cache line of data to be manipulated. For example, an entire line of memory may be loaded or stored into a buffer. In this way, the protocol processor is always in charge of data movement, but is freed from the burden of handling the individual data words. The parallel handling of data and control plays an important role in achieving efficient protocol processing in FLASH. We begin by discussing the control portions of MAGIC; later we describe the data logic and how the two interact.

Keeping memory request latency low and providing high throughput are essential requirements to achieving high system performance. In the FLASH design, protocol processing efficiency is therefore critical since all requests to memory and the network are serviced by MAGIC. The microarchitecture of MAGIC is designed for pipelined protocol processing at several levels to improve throughput. At the highest level, MAGIC provides a “macro-pipeline” that allows multiple messages to be in various phases of processing simultaneously. Figure 2.4 provides an overview of the three stages in the macro-pipeline. Some of these phases also include optimizations to allow the overall latency request to be reduced as well.

The first stage, the *Inbox*, designed by Mark Heinrich, prepares an incoming protocol message for processing by the PP. The Inbox selects requests from the three hardware interfaces (the main processor, network, and PCI IO bus), and two other sources (described below), the software queue, and the idle handler. The Inbox carries out the dispatch function described in Section 2.2.2, selecting the handler to be executed by the PP. In some cases it also initiates a speculative memory operation on behalf of the incoming message, for example reading the data from memory for a cache miss. By starting the memory fetch during the Inbox phase, before the handler is even able to execute, the overall latency to satisfy a request can be reduced. In addition, it removes the need to explicitly perform a memory request in the handler code, reducing total handler *occupancy* (the length of time the PP is busy to satisfy the request) [HHS+95].

For reads, the memory operation is speculative since it was started without first consulting the directory state. In the case of a cache miss, for example, the most current copy of the data may currently be in another node’s cache. In that case, the memory read was useless. In the case of writes, since our cache coherence protocol is ownership-based, a speculative write may always proceed without checking the directory state because only the current owner should issue a writeback. In Section 5.2.2 we describe how these “blind” writes slightly complicate the implementation of high performance coherent block transfer.

The Inbox also considers resource limitations in the system, such as full outgoing network FIFOs, and only selects requests from queues for which it knows MAGIC can provide certain minimum service guarantees. In Section 2.3.6, we describe how this feature is used to avoid deadlock in FLASH.

The second stage of the macro-pipeline is the Protocol Processor (PP) itself. The PP carries out the appropriate protocol actions, and often sends other requests to either the processor or the network. We discuss the protocol processor in more detail below. In the case of a cache miss, the protocol processor checks the directory state to locate the current data, and eventually sends the requested data to the processor.

The *Outbox* forms the third and simplest stage of the macro-pipeline by assuming control of any messages sent by the PP handler. The recipient unit of these messages cooperates with the Outbox and Data Buffer units to include the appropriate payload data associated with the message (if any).

2.3.2 The Protocol Processor

The Protocol Processor (PP) is a statically-scheduled dual-issue pipelined RISC processor. The PP was designed by Jules Bergmann. Its handler code and protocol state (such as directory state for cache coherence) is stored in a portion of main memory. To improve performance, the PP has its own instruction and data caches to make accesses to its memory more efficient.[5] In the initial implementation, the PP I-cache is 16 KB on chip; the PP D-cache is 1 MB off-chip. Both caches are direct-mapped with 128 byte lines.

The core instruction set of the PP is similar to that of the MIPS R3000 processor, extended to 64 bits. To improve performance, the PP provides several special instructions that enhance the performance of common protocol operations. For example, many protocols pack bits together to reduce the size of their state, so the PP provides operations to manipulate and query bits efficiently. Also, since the PP frequently interacts with the other units in MAGIC, a number of special instructions were added to access these units quickly. PP also supports a fast context switch capability that permits it to load the state for a new handler and begin executing it in only two cycles (pipelined). Appendix A illustrates the PP instruction set in more detail, including these extensions.

Though we added special features to optimize protocol processing, we also strived to keep the PP's design complexity to a reasonable level. To this end, the PP excludes many of the features found in general purpose RISC microprocessors that are less useful in our embedded environment. For example, there is no support for floating point operations or interrupts. Instead, the PP executes handlers to completion with no preemption. Furthermore, the PP provides no hardware support for

[5]Even though MAGIC can access all of the local main memory through its cache interface, data movement between external units is typically supported using the separate dedicated data paths in MAGIC. Thus, the MAGIC caches do not normally need to be kept coherent with the processor caches. The PP includes special instructions to provide this coherence if a specialized protocol should require it.

address translation (i.e., hardware TLB). Though address translation in hardware can be useful for protocols such as message passing, such support is not required by cache coherence protocols and could significantly complicate the PP implementation. Section 4.2 discusses this tradeoff and proposes efficient techniques that achieve similar functionality in software. Since the PP lacks sufficient protection mechanisms to support generic user-level code, we typically require protocol handlers that execute on the PP to be *trusted*. We consider ways to avoid this limitation in Section 7.1.2.

2.3.3 The Processor Interface

The Processor Interface (PI) is responsible for providing a high performance interface to the processor bus, while encapsulating it in the MAGIC message abstraction. When the processor makes a request on the bus, the PI queues a message for the Inbox containing the type of request and any associated data. Since the R10000 processor can support up to four outstanding requests per processor, the PI is also responsible for tracking these requests so that subsequent replies may be associated with the correct request. The PI was designed by David Ofelt.

The PI performs several critical functions that are invaluable for the protocols on MAGIC. When the PI receives a message from MAGIC for the processor, it must match the reply to its table of outstanding requests before passing it on to the processor. Part of this check determines whether any other coherence actions occurred for the line (such as an invalidation) while the request was outstanding. If a conflict between these actions arises, the PI uses a conversion table to determine what updated action to take. For example, in some cases an invalidation can arrive before a message providing the processor with its requested data. If this occurs, when the reply arrives it is passed on to the processor as a negative acknowledgement (causing the processor to retry). If the message is merely a reply to the processor request, the PI forms a bus command, launches the reply onto the bus, and then goes on to its next task.

The PI must also handle messages that are requests to the processor to access the second-level cache, called *interventions*. These requests are made by the PP to maintain coherence in the processor cache, extract data requested by another node, etc. Unlike the processor, which can have four outstanding requests, the PI only allows the protocol processor to have two outstanding interventions at any one time.[6]

The reply from a processor intervention comes in two portions. First the processor provides the result code called the *state reply*, which the PI exports in the PI Reply Register. The PI Reply Register is explained in more detail in Appendix A. Some time later, the processor provides the *data reply* (when appropriate) such as the line requested out of the cache. The data is streamed directly

[6]This restriction only applies to requests which require a status reply from the PI. We explain this issue in more detail in Appendix A.

into a data buffer, marking it full when finished. Since the state response arrives first, the PP can continue work while the PI fills the data buffer.

Processor and Cache Restrictions

The R10000 processor and its caches have several characteristics which impact key design decisions in the protocols we implement and study:

Speculation. Since the processor implements aggressive speculation support to improve memory system behavior, it may bring lines into its cache that are never actually used. As a result, references which emerge from the processor are not necessarily accessed by the control flow of the program (or modified, in the case of exclusive requests). In contrast, since uncached reads and writes may cause irreversible results outside the processor (such as I/O), they are never issued speculatively. This characteristic forces us to use uncached operations instead of cacheable ones in some circumstances where we must be certain that an address is actually accessed by the program's control flow.

Flushing lines. In some situations, the processor may wish to flush lines from its cache. To support this, the R10K provides a `cache` instruction. Unfortunately, this instruction is only accessible from privileged operation modes, thus it is impossible for code running at user level to reliably flush lines from the cache. As we describe later, this restriction forces us to utilize uncached operations to guarantee external visibility of certain operations issued within the processor.[7]

Pushing data into the cache. It is not possible to reply to the processor with data it has not requested. This prevents us from to implementing an update-based coherence protocol on FLASH, as well as a variety of other protocol optimizations that try to "push" data into the cache in anticipation of its use.

Accessing exclusively-held data. There is also no way to request exclusively-held data from the processor's cache and also leave the cache line in the exclusive state. In other words, "peeking" at a line of data held exclusively causes the processor to yield its ownership of the cache line. A subsequent write access by that processor requires an *upgrade* to an exclusive copy. In the message passing protocols we implement, for example, this restriction implies that copying data from the processor cache to another node necessarily perturbs at least the ownership state of the source node's cache.

[7]The only alternative would be to provide sufficient mappings that the application could force conflict replacements in its cache, but we felt this alternative was undesirable.

2.3.4 Data Buffers and Memory Access

MAGIC contains 16 *data buffers*, which are cache-line sized (128 byte) hardware buffers used to stage data through MAGIC in a pipelined fashion. These allow MAGIC to overlap the control processing and the data transfer associated with servicing a message. Whenever data is transferred into MAGIC from its interfaces, it goes directly into a data buffer without passing through the protocol processor. In most protocols, in fact, the data never passes through the protocol processor, but is transferred directly to the destination interface as directed by the PP. The data buffer allocator and status units were designed by Jeffrey Kuskin. The data buffer memory array was designed by Ron Ho and Evelina Yeung.

Buffer Allocation and Status Bits

Data buffers are assigned to a request, or *allocated*, by MAGIC when a new request is introduced into any of the interfaces: network, processor, or IO. The PP may also explicitly request that a buffer be allocated for its use if needed. Data buffers contain valid bits per-doubleword, which are similar in many respects to the full-empty bits found in architectures such as the Tera [ACC$^+$90], and others. These bits are cleared when a data buffer is allocated to a new request. As the buffer is filled with data, the valid bits are set when each doubleword arrives. If the buffer is consumed by another unit before it is completely filled, the valid bits cause the consumer to stall if it reaches a position in the buffer where the data is not ready. Using this approach, data can be aggressively pipelined from interface to interface with low latency.

The data buffers also have an associated state bit known as the *full bit*. The *full bit* can be used to override the individual per-doubleword valid bits and indicate that the buffer is full. The full bit is useful, for example, in situations where the PP composes the data in the buffer in software. Instead of setting sixteen individual per-word valid bits, the single full-bit can announce the buffer is ready for transmission.

Data buffers are referenced within MAGIC using their four-bit buffer number. In particular, the data buffer number is one of the fields in the MAGIC message header. When a request is allocated a data buffer, the message header that is formed includes the buffer's number. If the PP sends this header to another unit, the receiving unit consults the header to determine which buffer contains the associated data. In this way, the PP can easily tell the recipient of a message where to find its data without needing to handle the individual data words.

Buffer Usage

MAGIC provides the PP with several ways of manipulating data buffers. These include features to manipulate entire buffers as well as those to manipulate individual doublewords. The most common feature is the `send` instruction, which sends with its message the data buffer indicated in the message header, as described above.

One important usage of data buffers is efficient transfers to and from memory. As we described, the Inbox may issue a speculative memory read or write on behalf of an arriving request. This operation loads to or stores from the data buffer associated with that request. The PP can also explicitly load or store a buffer using its special `lblock` and `sblock` instructions . These operations can manipulate either the entire data buffer (128 bytes), or a single doubleword (8 bytes). Memory bandwidth is more effectively utilized by full buffer writes, but doubleword writes are useful for certain complicated reference patterns such as strided accesses for which a full buffer read would be primarily wasted.

In cache coherence protocols, cache lines are always moved as units. Thus, a data buffer fill from memory contains an aligned block of 128 bytes of data (i.e. data beginning at an address where the low seven bits of the address are zero).[8] Given this characteristic, restricting data buffer fills to aligned blocks in memory is reasonable for cache coherence protocols. However, this simplification is not appropriate for some message passing protocols where a message might be stored at a different cache-line offset than the one at which it originated. To address this limitation, MAGIC provides an additional buffer fill mode, called *double buffer* loads. These loads allow the PP to quickly fill cache-line *unaligned* data into its data buffers starting at any doubleword (64-bit) boundary. If this block of data is later stored, the effect is to shift the data *within the line* to a different offset than the one it originally had in memory. Because the architecture of FLASH supports communication protocols so well, this is the *only* hardware feature we added to explicitly support message passing. The memory controller and hardware support for double buffer operations was designed by David Nakahira.

Figure 2.6 shows how double buffer operations can be used to efficiently change memory alignment. Memory loads using the double buffer feature specify two data buffers (here we denote them A and B) and a starting doubleword offset. Buffer A is filled with an aligned block of data, but instead of filling from the first word in the data buffer, it begins at the provided offset. When the fill reaches the end of the buffer A, the remaining data of the line “wraps” and starts filling into the beginning of buffer B. By performing a second double buffer load of the next memory line, this time using the buffer B as the first buffer, the remainder of buffer B is filled in a similar manner. The

[8]The R10000 processor expects replies to cache misses to be in a special kind of critical-word-first ordering called *subblock* ordering. As a result, often the address transferred in the header does not actually end in seven zeroes, and the data in the data buffer is specially ordered to allows rapid restart from the miss. Though subblock ordering changes the *order* that the words appear in the data buffer, the words are still from the same aligned line in memory.

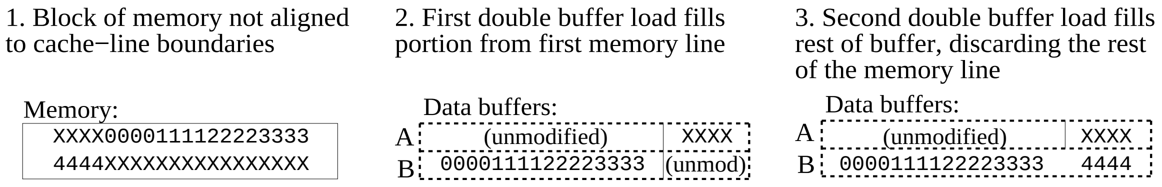

1. Block of memory not aligned to cache−line boundaries
Memory:
XXXX0000111122223333
4444XXXXXXXXXXXXXXXX
2. First double buffer load fills portion from first memory line
Data buffers:
A
(unmodified)
XXXX
B
0000111122223333
(unmod)
3. Second double buffer load fills rest of buffer, discarding the rest of the memory line
Data buffers:
A
(unmodified)
XXXX
B
0000111122223333
4444

Figure 2.6: Changing data alignment via double-buffer loads.

result is that buffer B is filled with data with a different cache line alignment than in memory. As an optimization, the double buffer mode can be selected but two identical buffer numbers provided. In that case, when the fill wraps, the remaining data words are simply discarded.

2.3.5 The Software Queue and Idle Handler

Besides the hardware interfaces from which requests can arrive for service, MAGIC provides two additional sources of requests called the *software queue* and the *idle handler*. The software queue is used by MAGIC to schedule handler invocation for itself at a later time. The idle handler is used to execute a handler periodically to perform a variety of maintenance tasks.

The Software Queue

For tasks that need to be invoked other than by the arrival of a message at a hardware interface, MAGIC provides the software queue. Its hardware support consists of a one-deep queue containing space for a message header, address, and handler PC. Since the queue is only one-deep, these are commonly referred to as the *software queue registers*. Unlike the hardware queues, which are associated with a particular external interface, the software queue registers are loaded explicitly by PP handlers. The element of the queue present in the software queue registers represents the request at the head of the queue; the remaining elements of the queue are stored in memory as described below. These registers are loaded with the headers of the message (as would be available if the message had come from any of the hardware interfaces) as well as a handler PC to execute to process the request. Once the registers are loaded, the Inbox can select and dispatch the software queue just as any other queue, except that the Inbox invokes the supplied handler PC instead of using the normal dispatch mechanism. To maintain the integrity of the software queue, any handler which executes from that queue is required to reload these registers with the next request before it completes.

The queue itself is maintained in software as a FIFO doubly circularly linked list in protocol processor memory and can consist of requests from different protocols since by convention all protocols use a common software queue header format. A set of short subroutines have been implemented that allow the user to add (`SWQSchedule`) and remove (`SWQUnschedule`) requests from the queue.

Once it has been on the software queue, a handler may also indicate that it is finished executing but would like to run again when its next turn comes around. A third routine (`SWQReschedule`) provides this behavior by yielding the PP in a controlled manner (reloading the software queue registers as required) but otherwise remaining on the queue. We differentiate these routines because the Reschedule routine is higher performance (since it does not need to manipulate the linked lists), so we use it whenever possible.

For intuition about the computation model the software queue provides, consider the parallel between the handlers on the software queue and the tasks of a very lightweight thread-based scheduler. In this metaphor, `SWQSchedule` parallels thread creation; `SWQUnschedule`, thread completion; and `SWQReschedule`, a thread yielding the processor to the next thread, but otherwise remaining active to run later.

Handlers executed from the software queue are different in two ways from handlers from the other sources. First, since the outgoing network queue needs of the next handler on the software queue are unknown, the Inbox conservatively schedules the software queue only when space is available on *both* the request and reply queues. Second, a data buffer is not allocated for handlers executed from the software queue. Instead, these handlers must explicitly allocate buffers with a request to the data buffer allocator. This decision was motivated by the observation that data buffer needs of software queue handlers cannot be predicted. Rather than assume a buffer is needed, when often they are not (consider the software queue handler that sends invalidations to a widely shared line), FLASH instead leaves the allocation to software. An artifact of this decision is that software queue handlers are not necessarily guaranteed to make forward progress. Our experience suggests that this does not lead to starvation or other performance problems in practice, but it remains an issue worthy of study with the actual hardware.

The Idle Handler

The idle handler is a single program counter value that is selected periodically by the Inbox.[9] Its purpose is to allow MAGIC to carry out a variety of maintenance functions from time to time, without using the software queue (which executes much more frequently). Among the tasks for which it might be used are the following: check for timeout of outstanding cache misses, update MAGIC performance statistics, verify system connectivity through “I’m alive” messages to neighbors, etc. In Section 6.1.6 we describe an additional use of the idle handler for assisting in timeout processing for the FLASH lock protocol.

[9]The term “idle” is actually a misnomer, since MAGIC executes this handler periodically whether it is idle or not. In fact, once its time arrives, it is the highest priority to be scheduled by the Inbox. Nonetheless, the functions it provides are typically associated with an idle handler in an operating system, thus the name has stuck.

2.3.6 Deadlock

Deadlock is a serious concern in design the FLASH system, as in any distributed system. In particular, the necessary conditions to generate deadlock are readily present in the FLASH network. To prevent this from ever occurring, the hardware and software in FLASH form a partnership, each component having its own responsibilities for deadlock avoidance.

The critical requirement in FLASH to avoid deadlock is that MAGIC must never send a message when outgoing queue space is not available, and cannot spin-wait for that queue space to appear. The reason for this is straightforward: if MAGIC did issue such a send, the PP would stall until queue space was available. However, the network could be full because another node is waiting to send to this node. Unfortunately, since the PP is non-preemptable, if it stalled in such a scenario it would stall forever, preventing the very condition by which the network could clear.

Avoiding this situation is accomplished in this way:

- The virtual lanes of the network are utilized as a Request and Reply network, similar to DASH [LLG$^+$90, Len92]. Messages sent on the reply lane are required to be *sinkable*—the protocol must have a way to accept the message without generating further traffic. Messages sent on the request lane are always permitted to send at least two reply messages. As a result, if the request cannot be satisfied, it can always send a single *NAK* to the requester. Since replies can always be satisfied and requests can always be turned into replies, network blockages can always be resolved.

- The Inbox establishes queue space guarantees for each of the incoming lanes of the network, and only schedules a handler from that lane when its queue space guarantees can be met. For example, the request lane must always be free to generate a reply, so requests can only be scheduled when the reply queue has outgoing space free. To be conservative, since the nature of the request is not known, the software queue can only be scheduled when *both* lanes have outgoing space.

- Handlers are required to obey the message sending restrictions of the queue on which they arrived. If handlers desire to send messages beyond their guaranteed privileges, they must first explicitly check the outgoing queue space (done with a `ls` instruction, see Figure A.1). If the queue space is present, the handler may send as many messages as the queue can hold.[10] If not, it must satisfy its request somehow and yield the PP without sending any messages.

[10]Since the macro-pipeline of FLASH processes several messages simultaneously, the Inbox may already have selected the next handler (and assured it the appropriate queue space). As a result, the currently executing handler can never *completely* exhaust the outgoing queues, but must always leave enough space for *one* handler which may be ready to execute next.

2.4 FLASH Software Environment

The design and implementation of protocols for FLASH entails a vast array of software support tools and simulators. These tools aid in the creation of protocols as well as the assessment of their performance during the design phase of FLASH. This section provides an overview of these tools to provide a stronger background for the environment in which these protocols were generated and evaluated.

2.4.1 Protocol Compilation and Scheduling

To make the design, implementation, and debugging of these complicated protocols more tractable, we chose to express the handler code in a high level language instead of assembly language. While this helps the protocol designer, it also requires that a range of tools be provided to help translate the high level language, in our case C, to machine code for the protocol processor.

Compilation

Rather than start from scratch, the FLASH team elected to re-target gcc, the GNU C compiler [Sta93] to generate PP assembly code. This was a good compiler base from which to start for two key reasons. First, since the PP was loosely based on the MIPS R3000 ISA, and since gcc was already ported to that ISA, a reasonably close starting point was immediately available. Second, since gcc is implemented using a flexible optimizer and code generator language designed for porting, the internals of the compiler were readily exposed. The FLASH port of gcc for the protocol processor, PPgcc, provides complete code generation features for protocol development, as well as a variety of peephole and code optimization features. These optimizations enable C-level constructs such as bitfields of structs and other efficient data structures common to protocols to be readily translated into the special bit manipulation support provided by the PP. In many cases, PPgcc-generated assembly code is efficient enough for the final machine. In other cases, it serves as a reliable staring ground from which hand optimizations can be applied. Gcc was ported to the protocol processor by Joel Baxter and Supratik Chakraborty.

Scheduling

In addition to generating the assembly language instructions themselves, FLASH adds another element to the code-development process since the PP is a statically-scheduled dual-issue processor. Unlike dynamically-scheduled processors such as the R10000, the PP requires the handler code to strictly match its asymmetrical issue restrictions, or else the instructions are decoded as illegal. Superscalar execution in the PP affords us increased instruction bandwidth per clock, which we find critical to achieve efficient protocol processing in software. The drawback to this feature is the need to convert the compiler-generated scalar-issue code to efficiently utilize the dual-issue capability.

Since PP instructions can be generated automatically from C-level protocol code, we chose to explore automated support for scheduling as well. We began with the Twine scheduler, written by Michael Smith as part of the Torch project at Stanford [Smi92]. This system was built using a former generation of the Stanford University Intermediate Format (SUIF) to internally store its code, relocation entries, and other information later needed to produce a valid object file [HAA$^+$96]. Twine was a perfect match for FLASH because the design of the PP was initially generated from a simplified version of Torch. Torch, like the PP, provides statically scheduled dual-issue, as well as support for instruction speculation. Though the speculation support was mostly removed from the PP, it still manifests itself in the PP's ability to provide simplified squashing branches. Twine, like PPgcc, is an effective staring point for our code. It optimizes most code well, though certain canonical sequences are poorly optimized warranting small hand optimizations.

2.4.2 FLASH System Simulation

We exclusively used simulation of a FLASH system during the development of the protocols in this thesis. The FLASH system was under design throughout this research, so the machine prototype was not available to test protocols. Simulation is a powerful technique to design protocols because it provides detailed visibility into the simulated machine.

We use the FlashLite system simulator, written by Mark Heinrich as part of the FLASH project. FlashLite models MAGIC and its network at the behavioral level, which provides nearly exact performance modeling, but not to the point of being cycle-by-cycle accurate. This allows us to use FlashLite to indicate FLASH performance, while avoiding the jump in complexity and maintenance to keep the simulator exactly in sync with the hardware design. The protocol processor (which is of particular interest in this study of protocol code) is simulated using a functional instruction emulator called PPsim. PPsim allows us to simulate the actual handler code and have the side-effects in PP instructions cause simulated MAGIC actions to occur in the FlashLite simulator.

FlashLite works in tandem with a CPU simulator which provides references from the processors in the simulated machine. Initially, FlashLite used the TangoLite reference generator, written in conjunction with the DASH project [GH93, Gol93]. Later, FlashLite was converted to work in conjunction with the SimOS simulation system, also developed at Stanford by a large team of researchers [RHWG95, Her98].

The SimOS framework provides the ability to boot and run a real operating system kernel under simulation, and to run multiprogramming workloads on top of the simulated kernel. The result is the ability to accurately characterize the performance of workloads that include operating system effects. SimOS models the simulated processors using two different CPU models: An R3000/R4000

CPU emulator called Mipsy and a high performance dynamic binary translation system called Embra [WR96]. In addition, we are able to use the Mipsy simulator outside of the rest of the SimOS environment to simulate applications while excluding OS effects, for the purpose of debugging, development, validation, and controlled performance evaluation. This combination of simulation tools has proven to be a powerful mechanism for simulating workloads on FLASH.

2.5 Summary

The FLASH project is focused on two main goals. First, studying the design of scalable multiprocessor systems to support the integration of cache-coherent shared memory and message passing. Second, to implement protocols in software to allow them to be easily corrected, optimized, or replaced altogether. In FLASH, hard-coded or table-based coherence protocols of systems like DASH are replaced by the MAGIC node controller. MAGIC contains an embedded protocol processor for control processing and dedicated data processing logic.

The FLASH system consists of an array of processing nodes, each containing a MIPS R10000 processor, a MAGIC chip, a portion of the machine's distributed memory, a PCI IO interface, and a port into the CrayLink Interconnect. MAGIC serves as the controller for communication within and between nodes, exchanging protocol information with other MAGIC chips using short messages. The instruction set of the Protocol Processor is based on the MIPS R3000 processor, then extended to improve performance of common protocol operations.

We study FLASH under simulation, using a detailed software model of its components. Protocol code for MAGIC is written in C and then compiled and optimized using a custom tool chain.

Chapter 3

Uses of Flexibility Beyond Cache Coherence

The centerpiece of the FLASH design is a node controller with embedded flexibility. One of the initial motivations for this design decision, as described in Section 2.1, is to enable cache coherence protocols to be implemented by software protocol handlers rather than hardware tables or finite state machines. The hardware units in FLASH operate under the control of these software handlers to assist in the data processing and protocol processing throughput in MAGIC. Experience gained from the design of FLASH and similar machines suggests that the implementation of cache coherence through flexible software handlers is a powerful and convenient way to design such protocols [KOH+94, HKO+94, RLW94, ACD+95, KCD+97].

However, the motivation for flexibility is not limited to the implementation of cache coherence protocols. Another goal of the FLASH project is to provide a single system that is able to efficiently support a range of communication protocols *beyond* cache coherence. This goal arises from the observation that multiprocessors of several different design styles have been converging to the point where they have practically the same hardware at their core. By designing a system with a small degree of additional flexibility, the result is a powerful machine that can support a range of models. These *alternate* protocols beyond cache coherence are the focus of this dissertation.

Since its conception the FLASH project has been interested in providing message passing support in addition to cache coherence. Historically, most large-scale parallel applications were implemented in the message passing style in the interests of scalability and performance. In recent years shared memory systems and applications have begun to provide a viable alternative; still, an ideological dichotomy remains in the high performance computing community between supporters of shared memory and message passing. In the past, systems were designed to support either message passing or shared memory, but not both; one key goal in FLASH is to demonstrate that a single system with a flexible engine can effectively support both message passing and shared memory with high performance.

Besides message passing, other protocols can also be built to exploit the flexibility in FLASH. One example is synchronization primitives such as locks and barriers. By providing support in MAGIC for these primitives, there is the potential to improve application performance by reducing the latency of these synchronization operations as compared to pure shared memory implementations.

This chapter provides an overview of the alternate protocols we study in the context of FLASH. We briefly explore the design space of these protocols, identifying the portion of the space on which we focus. In addition, we consider at a high level the individual software requirements of these protocols, determining the requirements they share. In the next chapter, we describe the implementation features common to multiple protocols (such as protection, interaction with the main processor, etc.), which serve as core mechanisms on which the alternate protocols are built. In subsequent chapters, we individually describe the implementation and performance of these alternate protocols. Chapter 7 considers other protocols that FLASH can support, but which are not studied in detail in this dissertation.

3.1 Block Transfer (Message Passing) in FLASH

In many early multiprocessors, message passing was the communication protocol provided to the user. In these systems, the memory on different nodes is completely separate, or *distributed*, and thus cannot be cached on remote nodes. Instead, for communication between nodes, applications send and receive data explicitly via *messages*.

The explicit nature of the communication in message passing systems affords certain advantages to the application. By crafting the communication manually, the programmer can closely manage and tune the movement of data around the system. A related benefit of explicit programmer control is that typically communication volume is reduced, since a message can be formed that contains only the *necessary* data.

Though it offers some advantages, explicit communication has its drawbacks as well. Programmer management of data transfer in message passing applications is not a simple task. At all times the application must keep track of the current copy of the data values and must establish a plan for transferring that data between nodes. In large applications, achieving this data manipulation correctly and efficiently can be very challenging. In addition, in applications where the data is largely shared, message passing programs may be forced to keep a copy of the shared data set in each node's memory, since caching of remote data is prohibited. In a shared memory system, the data would reside in one memory and be replicated in the *caches* of other nodes. These sources of complexity have traditionally made parallel programming in the message passing model much more cumbersome than in the in shared memory model.

Cache coherent shared memory addresses many of the concerns of the message passing model, but also sacrifices some of its advantages.[1] For example, shared memory eliminates the need to replicate the data set of the program since any node can access shared data. Furthermore, due to the cache coherence support the programmer never needs to explicitly manage the most up-to-date copy to achieve correctness.

On the other hand, shared memory hides interprocessor communication behind the load/store abstraction of normal programs, and usually communicates at a fixed cache-line granularity. This may ultimately reduce communication efficiency since the system, not the programmer, chooses which data to communicate. It also introduces the problem of *false-sharing*, artifactual communication that occurs because most protocols permit only one simultaneous writer per cache line. For example, consider the case when different nodes manipulate words on the same cache line. Even though these nodes may not read each other's results elsewhere on the line (and therefore no communication is inherently needed), cache coherence causes the entire line to move around the system.

3.1.1 Message Passing Overview

In a message passing machine, since messages are the only communication primitive, they are used for a range of functions. Messages may be used for synchronization and coordination (typically transporting a small amount of data) or for the transfer of large data regions, which have significantly larger payloads. In practice, messages may be used for a combination of these purposes since the delivery of a message usually implies some degree of synchronization.

In this section, we focus on the design space for message passing protocols that provide high performance block transfer. Our discussion considers the applicability of these protocols for FLASH, and identifies the common mechanisms which they share. Our goal is to demonstrate that block transfer message passing can be efficiently supported on FLASH.

Besides block transfer, small messages called *active messages* have been proposed as a generalized form of message passing that invokes computation at the receiver [DFK$^+$92, vECGS92]. Uses for active messages range in complexity from performing simple computation at destination nodes to more complex computation approaching the functionality of remote procedure calls. We consider support for active messages in Section 7.1.

[1]For brevity we refer to these simply as "shared memory" systems, but in all cases we are considering systems that have hardware cache coherence support.

3.1.2 Message Passing Models

Message passing abstractions exported to applications vary across a wide spectrum of functionality and semantics. Some protocols provide only basic data transfer capabilities and few other features. At the other extreme are more complex protocols, such as Intel NX [Pie88] and MPI [DOSW96, GLS]. These more advanced models provide such features as sophisticated buffer management, wildcards for receiving messages flexibly, and complicated multi-way communication primitives. We begin by describing this space to illustrate the kinds of protocols FLASH should be capable of supporting.

To better understand this spectrum of protocols, we consider two data points at the extremes, one model based on *connections* and the other based on *sends and receives*. The interesting conclusion to draw from these models is that despite the different communication features they export to applications, the requirements of their implementations are similar.

Simple Connection-Based Models

In a *connection-based* model, pairs of processes are associated, often statically, by *connections* (or *channels*). In this type of model, only the two "endpoint" processes can communicate using that connection. This restriction leads the application to open explicit channels between any pairs of processes desiring to communicate. In practice, if there are sufficient channels, they might be established early and remain open throughout the application. If there are relatively few, they might need to be time-multiplexed, i.e., assigned and reassigned as needed to allow different process pairs to use the limited resource. As the number of processes increases, the need for connections, which grows as $O(n^2)$, may become prohibitive.

A connection-based model has the advantage of simplicity for the system. Each endpoint can send data only to its communication partner, and this data is received at the other end in the order it was sent. Unfortunately, this simplicity makes a connection-based model challenging to utilize effectively in an application. For example, to receive messages out-of-order, multiple channels would need to be established between communicating peers (note that this also exacerbates the potential to exhaust the number of available connections).

Connection-based models enable the system to aggressively transfer the data to the receiver. This is made possible because connections and their associated buffers are established in advance, so each side usually knows if buffer space is available for it at its partner. If circular buffers are used, for example, all that is required is that the other end periodically indicate how much of the sent messages have been consumed.

Send/Receive Models

A more common communication model is based on *sends and receives*: NX, MPI, and many other protocols fit into this category [Pie88, GBD+94, DOSW96]. In this model, no static connections are established between communicating peers but instead each individual message send indicates its destination through some naming scheme. Likewise, a receive may specify a particular sender from which a message should be sought. The send/receive model is much more flexible since it allows any one of the application's processes to send to any other without advance set-up. It also has the advantage that processes can usually receive messages in a different order than they were sent.

On the other hand, this increased power also increases the complexity and overhead of the underlying implementation. The lack of connections may require more work on each individual message for authentication, buffering, or wildcard matching to determine its destination. In particular, allowing a process to receive its messages out of order significantly complicates the buffering implementation as compared to an in-order implementation. In some situations, if buffering has been preallocated, messages may be transmitted immediately (just as in the connection-oriented model). If preallocated buffers are exhausted, however, the sender must first negotiate with the receiver to avoid sending a message which cannot be accepted.

Common Requirements

Our goal in FLASH is not to provide a *particular* message passing model but rather to enable a *range* of protocols to operate efficiently. Fortunately, despite the differences between these protocols, we see that they have similar requirements. First, each requires a means to exchange control information with other nodes to arrange the buffering and transfer of large messages. Shared memory provides a convenient way to coordinate this communication between processors.

Second, for performance each requires a means to exchange data at high bandwidth. Once buffering has been arranged for messages, performing the actual transfer as quickly as possible is crucial to keep end-to-end message latency low. To move the application data, we turn to an accelerated block transfer protocol.

3.1.3 Providing Block Transfer

The most important decision to make at the outset is: where should we divide the functionality between MAGIC and the main processor? One approach would be to utilize the support of the MAGIC chip as much as possible. In other words, we could implement a specific message passing protocol in FLASH, one in which many of the higher-level functions of the protocol are provided by MAGIC.

However, when we increase the protocol functionality integrated into MAGIC, we encounter several major disadvantages. The first arises from the current MAGIC implementation. While MAGIC is capable of high performance protocol processing, its caches and dispatch table implementation limit the protocol code size which can achieve peak performance. In the current system with 16 KB MAGIC instruction caches, reducing overall protocol code size is critical to avoid instruction cache misses. In addition, though MAGIC provides the ability to dynamically change between protocols using the Inbox Jump Table, it is difficult for two large protocols to share that finite table efficiently. If the Jump Table size is exceeded, software is used to dispatch handlers that do not fit, increasing dispatch time significantly. The actual cost of dispatch depends on the position of the handler in the table; we have measured costs of 50 cycles, but they can be much higher.

The second disadvantage of increasing the functionality in MAGIC is that its protocols are harder to verify and harder to change than those implemented on the processor. In particular, verification is more difficult because errors in the protocol code are less visible and likely to be more harmful to the system than those in application-level libraries, since the former directly manipulates physical memory.

Finally, since all requests on the node require service from MAGIC, it is important to provide fair access to the protocol processor. As we increase the demands on protocol processing, system performance may degrade as the latency for servicing other processor requests increases [HHS$^+$95].

3.1.4 Integrated Shared Memory and Message Passing

Our goal in FLASH is to provide an integrated protocol which exports both shared memory and message passing features to applications. To achieve good performance and avoid the disadvantages described above, we provide a protocol which strives for high performance data movement but limits excess functionality. The interface we export to the application reflects the low level functionality we implement in MAGIC: accelerated *memory copy*. This protocol accepts a source and destination buffer address and performs the copy in MAGIC. We provide our support in that form for several different reasons:

First, shared memory allows more flexibility in how to perform inter-node data transfer. In a distributed memory machine the *receiver* of a message must decide where to store the message in its memory. While this may reduce the advance negotiation required between nodes, it would require MAGIC to implement a specific message passing policy. Instead, because of our shared memory environment, we are able to name the remote memory destination at the *sender*. The message passing library at the sender selects the destination buffer and indicates it to the sending node's MAGIC chip. With the decision of where to place the message having been made at a higher level, the receiving MAGIC chip can concern itself with high performance memory transfer and not with selecting an address for the data when it arrives.

Second, by providing an integrated solution, shared memory applications can utilize the same accelerated communication features, just as they would use other latency-hiding and bandwidth improvement techniques such as prefetching [MG91, Mow94] or DMA. This is possible not only because shared memory and message passing are allowed to interact, but more generally because the block transfer capability is not tied to a message passing protocol. In addition, the memory copy interface we choose is natural for both environments. Shared memory can use it directly, just as it would a normal `bcopy` call. Message passing libraries use memory copy to move the actual message data after selecting the buffer addresses.

Finally, by exporting the block transfer features directly to the main processor, we allow each library to utilize the low-level features as efficiently as their abstraction permits. For example, a simple connection-based model designed for use in a simple producer-consumer application could be implemented directly on the memory copy interface. In contrast, the additional support provided by an MPI library might achieve somewhat lower performance—but without affecting applications using lighter models. If instead we had chosen one particular model to implement in MAGIC, other models with different requirements would be forced to build on top of it even if their abstractions were poorly matched.

3.2 Synchronization Primitives

Parallel applications implemented in the shared memory style are often required to synchronize between different processes to ensure correct execution. Synchronization operations can be used in a variety of ways, for example:

- Provide mutual exclusion to data structures.
- Distribute work between processes.
- Ensure that processes proceed through application phases at the same time.

Providing good synchronization performance is critical for achieving scalability in parallel systems. Inefficient synchronization increases the cost of parallelism, since processors waste time synchronizing when they could instead be doing useful work. More generally, if synchronization is expensive or inefficient, applications are forced to cooperate at larger granularity to amortize synchronization overhead. If synchronization cost can be reduced, it enables applications to seek out finer granularity parallelism that may ultimately increase performance.

In FLASH, we can leverage the flexibility of MAGIC to accelerate synchronization primitives through specialized protocol support. This section describes a variety of synchronization primitives and their application uses and requirements. In Chapter 6, we describe the implementation of these synchronization primitives on FLASH.

3.2.1 Locks

One of the most common synchronization primitives, from which most others can be generated, is the *lock*. At any one time a lock can be held by only one processor, and so it can be used in circumstances where mutual exclusion is required. Shared memory locks perform very well in some circumstances. In particular, when access to a lock is not contended, shared memory can achieve low latency since lock requests succeed immediately and do not need to retry. Since shared memory locks are comprised of normal memory locations, a lock is cached when acquired. As a result, if contention is low and the lock is requested again soon, the lock might be found in the cache and acquired at very low latency.

When these scenarios are not met, however, the performance of shared memory locks degrades. For example, as contention increases, the need for exclusive ownership to modify the lock may cause requests to be negatively acknowledged, or lines to ping around the system. In degenerate cases, a processor that is granted ownership of the lock may have the lock snatched away before the lock can be acquired—this scenario can lead to livelock if steps are not taken to guarantee forward progress.

Furthermore, since ownership of a line is acquired based on which requester reaches the home node first, there is no guarantee of any fairness or ordering of lock acquisitions. In particular, when a lock is held the other requesters all acquire shared copies and spin while the lock is unavailable. When the lock holder releases the lock, the home invalidates all the sharers—causing them to re-fetch the line in an effort to acquire the lock. This results in a rush of requests for the line in quick succession, even though only one of the waiters ultimately succeeds in acquiring ownership (and the lock).

Our approach to providing locks in FLASH addresses the weaknesses of locks implemented on normal shared memory. Since shared memory locks perform well in low contention cases, additional protocol support would be of no benefit in that regime. Instead, we focus on accelerating high contention locks through a variety of techniques. First, we address the fairness issue caused by the rush of requests that follows an unlock of a contended lock. Instead of using pure shared memory, requests for these locks are queued so when a lock is released only the head of the queue is notified. A similar approach was provided by the DASH system to optimize lock variables [Len92], and in software by Mellor-Crummey and Scott [MCS91a, MCS91b]. Second, we optimize the transfer of a lock between holders, avoiding the extra traffic and latency introduced by the shared memory model. Third, we try also to provide the advantages of shared memory, including efficient repeated acquisition of a cached lock.

3.2.2 Barriers

Another important synchronization primitive for parallel applications is the *barrier*. Barriers force a rendezvous of all processors, i.e., when a processor reaches the barrier it must wait until all others arrive as well, and only then can all proceed. Barriers are typically used in phase-based applications, so that each processor completes its work in the current phase before any can enter the next one. This prevents subsequent phases from mistakenly reading incomplete results from other processors if they are still working on previous phases [WOT+95].

Barriers are normally implemented in the parallel macro package or communication library of a parallel computer. Their implementation often consists of a single data structure that stores the count of processors that are currently waiting. In that implementation, each processor in the application needs to update the shared data structure, resulting in $O(n)$ best-case performance for the barrier. Just as in locks, contention for the barrier data structure can decrease performance significantly. Such contention may be high if processors arrive at the barrier near each other, which is increasingly likely as machines scale.

A more efficient barrier implementation would utilize the parallelism in the machine to its advantage, using a tree-based structure instead [MCS91a, MCS91b]. This improves the barrier's best-case complexity to $O(\log n)$. A tree barrier can be implemented in the application library just as a normal barrier, or can be supported using a custom protocol.

Barriers may further benefit from support in MAGIC since each level in the barrier tree can be traversed with lower latency. First, MAGIC is closer to the network than the processor, so communication with remote nodes occurs as quickly as FLASH can support. Second, shared memory must go through the coherence protocol to communicate with remote nodes. An application-based tree barrier would acquire ownership of the next tree node and perform its modifications in its cache; the next barrier stage would then just acquire it back. MAGIC can avoid this unnecessary communication and advance to the next stage of the barrier directly, using specialized techniques we describe later.

3.3 Summary

This chapter provided an overview of the alternate protocols we study in this dissertation. Considering the protocols first at a high level allows us to identify the key requirements they share. The next chapter builds on this overview to support these shared needs; later chapters consider the implementation of the individual protocols in more detail.

The first major protocol we study is message passing. Providing a message passing model has both advantages and disadvantages to the programmer and the system. We described a range of message passing support before selecting a fundamental memory copy primitive for FLASH from which more complicated protocols can be constructed.

We also described synchronization primitives: locks and barriers. Providing efficient synchronization is critical to performance in parallel systems. Our goal will be to improve synchronization performance over conventional shared memory techniques using the support provided by the FLASH system.

Chapter 4

Alternate Protocol Fundamentals

In the previous chapter, we identified communication and synchronization functionality that may benefit from implementation in MAGIC. Even though these various protocols perform very different functions, they share several fundamental characteristics. Before examining the implementation of these particular alternate protocols in detail, this chapter describes the fundamental *mechanisms* they share [HGG94, HGDG94]. By "mechanisms" we refer to both actual protocol *code* and algorithmic *techniques* to accomplish certain critical actions within alternate protocols.

One widely required mechanism is the ability to initiate alternate protocol operations on MAGIC under control of the main processor. This requires an efficient and powerful interface between the processor and MAGIC. In many respects the processor considers MAGIC to be a memory-mapped device, i.e., it can issue commands through accesses to special memory locations. However, in the interest of performance and protection, our implementation extends this interface in several ways. We describe a family of processor/MAGIC communication techniques that are used by our protocols. These techniques enable communication in either direction and with varying degrees of functionality and performance. We then extend this basic interface to show how it can be used in the context of virtual memory. Since our environment contains a full-fledged operating system, correctly handling virtual memory and assuring memory protection is of critical importance to allow widespread use of our protocols.

Once these operations have been initiated, MAGIC's tasks are centered around manipulating user data. This would be straightforward in a distributed memory system where data can only reside in local memory or the local processor's cache. In our shared memory environment, however, coherence is much more difficult and often entails communication with remote nodes. In general, providing coherence for our alternate protocols requires a subset of the functionality in the cache coherence protocol. In some cases, custom coherence support is integrated into the alternate protocol; we describe an example of such support in Chapter 5. In this chapter we describe instead a shared technique for coherently modifying user data that leverages a corner case of the cache coherence protocol. This technique, called *PP ownership*, reduces protocol code replication as compared to custom coherence support, in exchange for reduced performance.

Section 4.1 describes the processor/MAGIC interface for basic command sequences, then Section 4.2 extends the interface to operate in a virtual memory environment. Section 4.3 provides a brief overview of protection features in MAGIC, though many protection features are protocol-specific and are addressed later. Finally, Section 4.4 explains the PP ownership technique for coherently modifying data in the PP.

4.1 Processor/MAGIC Communication

The ability to leverage MAGIC to provide computational functionality outside the compute processor is a powerful feature of the FLASH system. To utilize this capability, we must provide a processor interface to MAGIC for initiating protocol operations in an efficient, atomic, and protected manner. Similarly, some protocols running in MAGIC require service from the processor, so a related interface is needed to cleanly request the processor's attention. In this section we address the interface between the processor and MAGIC which provides these features. Since the requirements of each alternate protocol are somewhat different, we describe several techniques that offer varying tradeoffs of functionality and performance. Though the details of the techniques we propose are specific to FLASH, many of the concepts apply to other systems with similar characteristics [RLW94, BLA+94, MKAK94, NAB+95, LC96].

As described in Section 2.3.3, MAGIC's processor interface provides a highly optimized abstraction of processor bus operations. Our interface builds on top of the existing PI features to provide the additional functionality needed to initiate alternate protocols. At a high level, we find the goals for initiating alternate protocols are similar to those needed to invoke an RPC [SPG91].

- The ability to name an operation for MAGIC to execute.
- The ability to provide multiple arguments to the operation (including both data values and memory addresses).
- Guaranteed atomicity (and an assurance that an operation is executed "at most once") despite preemptive multiprogramming on the processor.
- A means to receive one or more result values from the operation which was invoked.

In other systems, powerful features such as those available in MAGIC are typically exported through a system call interface, and are thus protected by the operating system. Unfortunately, the high overhead of system calls leads us to strive for an interface provided directly at user level. The challenge is to achieve such an interface without subverting the protection features normally provided by a system call. These protection features come in several different forms:

- The OS prevents applications from using resources to which it does not have access (e.g., applications cannot access certain devices unless they have special privileges). In our case, this protection corresponds to our restricting alternate protocols to applications authorized to use them.
- The OS uses virtual memory to protect and share the physical memory of the machine. As a result, communication from user-level applications necessarily describes memory in virtual addresses. Special steps must be taken to convert these to the physical addresses used by MAGIC.
- System call arguments are checked to be sure that the services are only used in approved ways. Examples of this include reading beyond the end of files, trying to open invalid devices, and trying to use system calls reserved for privileged users. The same kinds of argument checking must be done for protocol actions initiated on MAGIC.

The following sections describe how alternate protocols are initiated on MAGIC in a way that satisfies all these requirements. We call this interface Protocol Processor Calls (PPCs), and Protocol Processor Registers (PPRs), since they are intended to be somewhat analogous to the RPC functionality common to many systems. Section 4.1.1 describes how command sequences are used to initiate PPCs from the processor. Section 4.1.2 describes a simplified version of a PPC, the PPR, that is designed for higher performance but more limited functionality. For communication from MAGIC to the processor, we provide a mechanism called an OSPC, explained in Section 4.1.3.

4.1.1 The Protocol Processor Call (PPC)

To allow the processor to initiate alternate protocols on MAGIC, we provide an interface known as a Protocol Processor Call (PPC). A PPC consists of a series of commands with a controlled format, through which the processor can specify the operation it wants MAGIC to perform. The PPC interface is deliberately designed to be fairly similar to the RPC style of communication common in many operating systems and distributed systems. The analogy is straightforward: PPCs are a request for service from MAGIC, just as most RPCs are a request for service from another system, node, or kernel. PPCs, like RPCs, may run quickly and reply inline, or they may take longer and return a result asynchronously. This section describes the bare essentials of the PPC interface, illustrating how it can be used to initiate simple alternate protocols. Subsequent sections extend the basic PPC definition given here by explaining how they interact with virtual memory, and multiprogramming.

Memory-Mapped Commands

PPCs are issued by the processor through a memory-mapped interface to MAGIC. To implement this interface, we leverage the flexibility of MAGIC to interpret processor bus requests any way it chooses. We “overload” uncached writes by the processor to specify to MAGIC either a command or argument. Similarly, we use processor uncached reads to request a data word from MAGIC (a result or return value). Uncached mappings also prevent processor speculative references from confusing the sequence or introducing unintended references into it. This problem arises because the R10000 processor aggressively generates speculative references in an attempt to hide memory latency. Fortunately, the R10000 cannot speculatively issue uncached references since they may be permanent (e.g., if the address corresponds to an IO device).

To enable this interface, but also allow normal uncached operations by the processor, we must distinguish PPC commands from normal accesses in some way. In other words, when MAGIC detects a distinguished PPC reference, it must know to interpret it specially and not to merely carry out the memory operation it seems to indicate.

The FLASH system provides two features we can use to distinguish address ranges as special. These two features are each useful in different situations; later we describe protocols illustrating their respective uses. Both features are accessed through the memory management unit support already provided by the processor. The key benefit of leveraging the memory management unit is that the operating system can expose the memory mapped interface directly to user programs without compromising protection or requiring system calls on each use. The DASH system used similar methods of distinguishing memory references for its alternative memory operations [Len92].

The first feature for distinguishing PPC references is provided by the R10000 processor, which allows several *uncached attribute* or “flavor” bits to be associated with uncached mappings in the TLB. These bits are exported as part of the bus command when an uncached operation using that TLB entry is generated, and are among the fields used by the Inbox in dispatching the appropriate handler for a request. Thus, the Inbox can use the flavor bits to select a special handler, in this case one which interprets the access as a memory mapped command to MAGIC.

The second feature is built into MAGIC, which artificially divides the physical address space into four distinct regions, called *address spaces* or simply *spaces*. The address format is illustrated in Figure 4.1. Accesses to different spaces do not specify different memory locations, but instead specify four different names for the *same* memory location. Space bits, like flavor bits, are used by

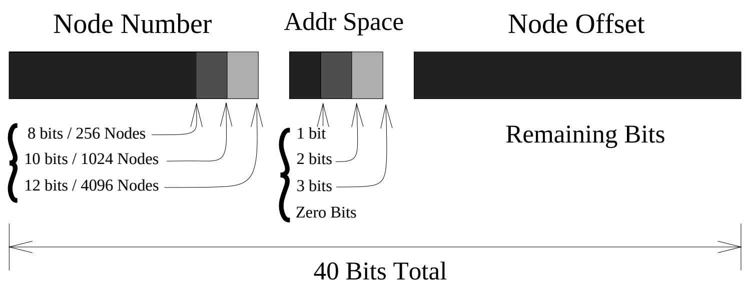


Figure 4.1: FLASH Address format. FLASH allows variable sized node number and address space fields. The remaining bits are assigned to form the offset into the local node's memory, though this memory need not be fully populated with DRAM.

the Inbox as one of the criteria for selecting a handler, and thus allow us to distinguish references similarly.[1]

PPC Initiation and Atomicity

A PPC consists of a series of uncached references that, using one of the techniques described above, emerge from the processor with distinguishing characteristics. This sequence normally consists of a series of writes specifying command and arguments, followed by a read that requests a success or failure indication. As each command reference is emitted from the processor, the PP handler logs the address and data value of the reference into protocol memory. At the end of the sequence, the terminating command cues MAGIC to interpret the commands and arguments it has received.

As we described, the PPC specifies its command using a memory mapped interface. In that interface, the *address* referenced indicates the command the processor wishes to invoke; the *data values* written to those addresses are the arguments the processor provides. The command addresses are arranged so that each different PPC request type is allocated a different cache line in the memory mapped command region. Doubleword offsets within that line are considered sequence numbers for the arguments pertaining to that command. Thus, PPCs are issued as a series of writes to sequential offsets (with exceptions to this rule in some cases, as described later), which allows MAGIC to verify that the commands are in sequence.

The last memory mapped operation in a command sequence is usually an uncached read. This read performs two functions: it notifies MAGIC that the command sequence is complete and should be committed, and it requests a response to indicate if the PPC has been accepted.[2] A negative

[1]Unlike flavor bits, which can only be used for uncached accesses, space bits can also be used for cacheable accesses. Though the cacheable aspect of address spaces is not useful for implementing PPCs, it allows us to select a different coherence algorithm depending on the space that is referenced. In Sections 6.1 and 6.2 we describe how custom coherence algorithms can be used to specially optimize the communication between the processor and MAGIC for PP-supported synchronization protocols.

[2]This does not imply the completion of the requested operation. Completion is usually in other means: coherent memory locations or subsequent PPCs or PPRs.

acknowledgement indicates that MAGIC has rejected the sequence (either due to temporary resource limitations or errors in the commands). Command sequences that fail may be retried by the processor if desired.

Unfortunately, sequence checking alone is not enough to guarantee that the commands arrive correctly at MAGIC. For any PPCs initiated at user level, the sequence of commands may be interrupted at any time by a context switch, interrupt, or exception. Though unlikely, it is possible that two different processes could be in the middle of initiating the same PPC. If this happened, one process might issue a command that gets mistakenly interpreted as part of the other process's PPC. We can solve this problem and make the PPC initiation mechanism robust by flushing any incomplete command sequences when context switches occur on the main processor. To accomplish this, we reserve a PPR (described below) for the kernel to notify MAGIC when context switches occur.[3] Later, when the process resumes, the PPC fails since its references are of sequence; it can then be retried.

Compared to the solution used in CM-5 [Thi92] and Alewife [KA93], our approach does not require saving and restoring of the commands across interrupts. In addition, the programmability of MAGIC allows us to customize the command sequence protocol for various uses as opposed to providing a single hardwired protocol [ALK$^+$91, LLG$^+$90]. While the above approach cannot absolutely guarantee the forward progress of a PPC that is repeatedly interrupted, it is unlikely to be interrupted indefinitely. If in practice frequent interrupts turn out to impede forward progress, we can instead use a solution that maintains separate command queues per process inside MAGIC. While this would increase the overhead of the context switch handler, it would prevent PPCs from failing due to sequence interruptions, since the sequence could be resumed where it left off.

PPC Replies to the Processor

Recall that following the sequence of commands forming the PPC, an uncached read is issued to complete the sequence. Depending on the service invoked by the PPC, the reply to the uncached read can be used in several different ways. We briefly consider the characteristics of these different approaches:

Two-Phase Initiation The two-phase initiation style described above is the most common. In that style, the result code from the PPC refers only to the acceptance of the command sequence itself; the operation on MAGIC completes asynchronously. If desired, the processor can later query the result of the operation using one of several techniques. First, the PP can coherently update a location (or locations) in user memory which the processor can read. The processor

[3]Since context switch notifications are relatively infrequent and the handler it invokes is very short, the overhead introduced is small. In the case of clock interrupts and other more temporary interruptions to user processing, the kernel only needs to notify MAGIC before itself invoking PPC services, or before other user processes can run.

could also issue a subsequent PPC or PPR, the entire purpose of which is to poll for the completion of the asynchronous operation. In most cases, coherent memory updates are the most efficient technique.

Immediate Response In some cases, MAGIC can carry out the action requested by the PPC immediately. For those operations, the return value can be more significant and actually indicate that the requested operation is complete. Since the processor is stalled during this operation waiting on the result of its uncached read, this style of reply should only be used for operations which complete in a fairly short time. Initializing MAGIC state records or consulting performance monitoring statistics are examples of PPCs which can provide an immediate response.

4.1.2 The Protocol Processor Register (PPR)

The PPC mechanism is most appropriate for invoking complex MAGIC services. Typically these services accept multiple arguments and then return a status reply since initiation may fail due to resource limitations. However, there are many requests that can *always* be accepted by MAGIC when they arrive and can be guaranteed to always complete. These characteristics permit initiation to be simpler and faster and not require a status reply. For these types of requests, we designed an optimized PPC, known as a PPR or *Protocol Processor Register*. This interface is named because PPRs logically act as a register in MAGIC that can only be read or written, even though in reality they execute a handler, just as a PPC.

PPRs are accessed through a command space uncached read or write. These individual accesses are the same as those for PPCs, but to a different portion of the command region. However, unlike a PPC which consists of a sequence of commands, a PPR is accessed with only a single read or write. This simplified interface has two attractive characteristics (for the cases when it can be used): *(i)* since it consists of only a single command, it is automatically atomic; *(ii)* a PPR write allows the processor to proceed immediately without waiting for MAGIC.

PPRs may be used in two main ways. The simpler version is the more common: the PPR is implemented in MAGIC by just reading or writing a memory location corresponding to the PPR. In this configuration, the PPRs backing memory is usually accessed through the MAGIC data cache instead of the normal processor memory mechanism (In other words, the PPR value is part of the protocol data itself). This allows the protocol executing on MAGIC to read the value of any PPR registers without checking for coherence with memory. In fact, this is one of the most useful applications of PPRs, to configure machine parameters used by the MAGIC protocol. Some interesting examples of this include: reading performance monitoring registers, querying MAGIC's position in the network, manipulating the interrupt masking capability of the processor interface.

PPRs may also execute a handler, similar to a PPC, which performs some processing for the request. Essentially, PPR writes are single argument PPCs returning no value; PPR reads are PPCs taking no arguments and returning a single value. For example, when the processor context switches, a PPR is sufficient to notify MAGIC of the new context. MAGIC not only notes the change, but also takes steps such as flushing partially formed PPCs. An example of a read PPR is one that examines the hardware units in MAGIC and returns a bit vector to the processor representing which of the units have experienced errors.

4.1.3 The Operating System Procedure Call (OSPC)

Just as the PPC mechanism is used by the processor to request services of the PP, we also provide a second mechanism, called an OSPC or *Operating System Procedure Call*, used by the PP to request services of the operating system. OSPCs have a number of important applications in the FLASH system. As we describe in 7.1.1, OSPCs are used to support processor-implemented active messages by efficiently interrupting the processor to execute the requested handler. They are also used within the operating system for interprocessor communication between the kernels of two different nodes. The Hive operating system relies on the OSPC mechanism as part of the fast messaging support it uses to maintain reliability [CRD$^+$95]. Finally, the PP itself can use OSPCs to request attention from the operating system. For example, in Section 4.2 we describe how the PP occasionally needs to request virtual to physical translations from the processor. The OSPC was initially developed jointly with John Chapin, one of the members of the Hive team.

Base OSPC implementation

An OSPC is a request to the processor consisting of a request type and a number of arguments. In the basic implementation of OSPCs, each request is one cache line long: one doubleword for the type and up to fifteen arguments. Since multiple OSPCs may be outstanding at one time, we store the pending requests in a circular queue in MAGIC protocol memory. We currently implement two separate queues to provide two priority levels. When a request for an OSPC arrives (or is generated internally by the PP), first the OSPC request is formed and stored in PP memory. Then a processor interrupt is asserted to indicate an OSPC is waiting.

When the processor services the interrupt, it must read the request at the head of the queue to determine what action to take. Ideally, for efficiency we also allow the processor to cache the OSPC request it reads. However, since we implement the OSPC queue in MAGIC protocol memory, normal cacheable reads are unable to access this memory. This is by design, since it eliminates the need to maintain coherence with the processor cache when composing the OSPC request. Instead, the processor maps a page in *cacheable noncoherent* mode, and reads a reserved address from that page.

MAGIC intercepts the reserved address for accesses of that mode and supplies the OSPC request at the head of the queue by filling the data buffer from its protocol memory.

The processor can then service the request as needed, referring to the data arguments efficiently in its cache. In the midst of processing the request, cache conflicts may cause the OSPC request to be replaced from the cache prematurely. If the processor reads the request address again, MAGIC provides the same request to satisfy the miss. Eventually, when processing of the request is complete, the processor advances the queue. It does this by sending an acknowledgement to MAGIC (a PPR), and invalidating the request line from its cache. From the acknowledgement, MAGIC knows it is safe to advance the queue, and de-asserts interrupts if the queue is empty.

There are two alternatives for how to check to see if additional OSPCs are waiting. The kernel could read the reserved address again and process additional requests it receives until it finally receives a reply indicating "no OSPC waiting". It could instead pause slightly (to allow MAGIC time to modify the interrupt mask), and return from the interrupt. If other OSPCs follow, it would take the interrupt again and process it as before.[4] The former approach is more efficient if multiple OSPCs tend to come at once since it avoids unneeded interrupts; the latter is more efficient if OSPCs tend to arrive individually, since extra processor/MAGIC round-trips can be averted.

OSPC Optimizations

The basic implementation of OSPCs described above has a number of minor performance problems. Below we describe these problems and explain optimizations that can be applied to improve OSPC handling performance and reduce overheads. These optimizations are particularly important in the context of latency-critical uses of OSPCs, such as interprocess kernel communication.

Reducing OSPC request size. Most requests do not need the full cache-line of arguments described in the base implementation. Unneeded words consume time to fill from memory and travel on the bus. For OSPC requests with only one or two arguments, it is more efficient to reserve a special kind of OSPC which the processor accesses with uncached reads. Since uncached reads are reliable, it also eliminates the need to acknowledge OSPCs explicitly—the ACK is implicit when the last word is read. The MAGIC implementation is sufficiently optimized for moving cache lines that this approach can only show gains over the base approach for OSPCs of 1–2 words.

Avoiding unneeded trips to memory. In the base implementation, OSPC requests were always written to and supplied from memory. We can instead choose to store the OSPC request for the head of the queue in a MAGIC data buffer. We do this by leaving the data buffer allocated

[4]It may also take an unneeded interrupt if MAGIC has not had the opportunity to de-assert the interrupt. This does not represent a correctness problem, it merely causes wasted processing.

after the initial request handler finishes. This allows the read miss handler to supply the OSPC request *immediately*. This optimization must be applied carefully since it restricts the availability of data buffers, but since only the head OSPC request is stored in this way, it poses no risk in most situations. Even though we hold the request in a buffer until the processor asks for it, we also write it to the circular queue in memory so that we can provide the request again if it is prematurely replaced from the processor cache.

Reducing acknowledgement overhead. When multiple OSPCs arrive in succession, the processor sends an acknowledgement to MAGIC when it finishes with the head request, so the circular queue can be advanced. It follows with a read miss to the reserved location to access the subsequent request. It is not fundamental that these two handlers are separate. Instead, we could use a "double buffering" approach in which two *different* reserved addresses are provided. When a processor changes from one address to the other, it is implicitly acknowledging the first request and at the same time asking for the following one. By putting both functions in a single handler, the total overhead can be reduced.

4.2 Virtual Memory

Software services provided on MAGIC are usually oriented around manipulating memory in a special way on behalf of the processor. Memory access by MAGIC occurs exclusively at the physical address level, since MAGIC is effectively the memory controller for the physical memory devices. In contrast, user-level processes can only specify virtual addresses. This introduces the need to perform virtual to physical translation as some part of the PPC mechanism. Of course, since the kernel assigns virtual to physical mappings, a system call is sufficient to authentically translate addresses from the user for use in MAGIC protocols. Unfortunately, system calls are expensive, so the challenge is to achieve this same functionality without system calls, through purely user-level accesses.

In this section, we describe our approach to allowing the user to communicate *authentic* translations to MAGIC at user level, within the confines of the PPC mechanism. There are two problems to be solved to ensure that MAGIC uses authentic physical addresses:

How can authentic physical addresses be generated from user level? Our goal is to allow the user to provide not only simple integer arguments as part of a PPC, but also addresses (i.e., pointers to data to be acted upon). We describe two approaches for providing authentic physical addresses to MAGIC, one based on translations using the processor's MMU, the other using the PP to translate addresses in software.

How can we guarantee that these translations remain valid while MAGIC is using them?

We show three different techniques that may be used to ensure the validity of these addresses while in use by MAGIC. Our underlying assumption is that changes in virtual to physical translation mappings of pages involved in the transfer are relatively infrequent (since they are currently being used, they are less likely than other pages to be victimized). Therefore, our general philosophy is to provide the necessary mechanisms to handle translation changes without slowing down the case where translations do not change. We allude to similar techniques which would be preferred if translation changes were more frequent.

4.2.1 Providing Authentic Translations from User Level

As discussed in Section 2.3.2, one of the simplifications in the design of the protocol processor (PP) as compared to a general microprocessor is the removal of the TLB. We made this simplification because *(i)* cache-coherent memory operations do not require address translation since the processor already presents MAGIC with physical addresses; *(ii)* handling TLB and page faults on the PP would add significant complexity; and *(iii)* a small hardware TLB may not be an effective structure for caching translations, especially since PP reference patterns are different from that of code executed on the compute processor.

Instead, we support the required address translation for protocol operations using other techniques. We present two techniques that accomplish this task. The first uses a special mapping of memory exported to the user, through which addresses may be referenced as part of a command sequence. The second passes virtual addresses down as part of the PPC, then uses a software TLB to generate an authentic translation.

Providing Translations Through a Shadow Mapping

The first technique is based on the observation that user processes are *already* continually providing authentic addresses to the system via the memory management unit in the processor. Since all this hardware and software support is already present in the system, it begs the question: Why not simply leverage that support to allow the user to generate *addresses* for command sequences as well? For example, a naive approach might be for the user to merely reference the appropriate address in the middle of the command sequence. This would result in the desired translation on the bus (assuming the line is not cached).

Processor translation *is* sufficient to be used as part of a command sequence, but it must be utilized in a more sophisticated way than the example above. The first problem is that the line may indeed be cached, and then the reference would not emerge on the bus. In fact, since this address is the subject of a PPC, it is quite likely to have been recently referenced and thus cached. To ensure visibility of processor references of this type, we are forced to provide a separate uncached mapping

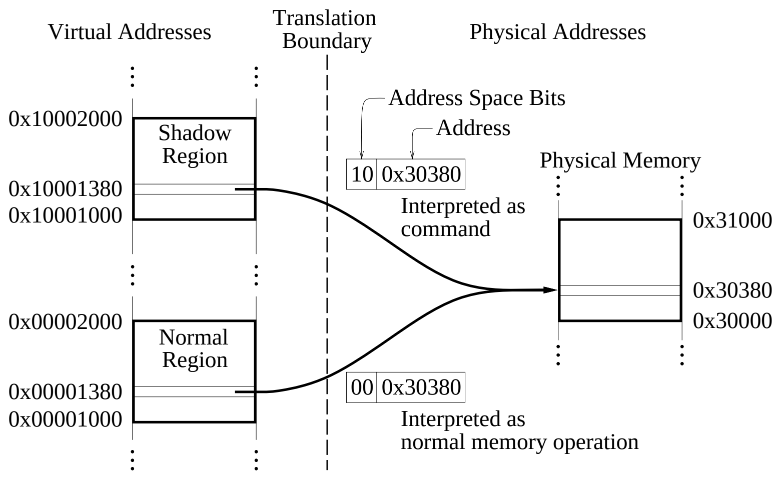


Figure 4.2: Example of double mapping illustrating the use of alternate physical address spaces.

of the page, as before. The second issue is that, similar to PPC references, we must distinguish these references so MAGIC can interpret them specially.

We provide the user with an uncached mapping to ensure visibility, and use the techniques described in 4.1.1 to distinguish its references. We call this a *shadow mapping*, since we provide the user with two mappings for every memory page, as illustrated in Figure 4.2. The OS provides a shadow mapping by exporting a shadow region which has a translation only differing in its address space (or flavor). References to the normal region are interpreted as ordinary memory operations while references to the shadow region are interpreted as part of the PPC, providing MAGIC with authentic physical addresses. Recall that in a PPC command reference, the processor writes an argument value to a fixed command location. In contrast, to provide an address as part of a PPC, the *address* which is referenced indicates a pointer that is part of the PPC. Note that a single user-level reference to the shadow region provides MAGIC with an authentic translation usable for a whole physical page.

In the discussion of PPCs, we described how sequence number checking required us to restrict PPC command space references to consecutive addresses. This allowed us to detect when references arrive out of order or when a sequence is interrupted. Though address references using the shadow mapping violate this restriction, the same sequence number guarantees can still be assured by extending the rules to accommodate address references:

- PPCs are required to begin with a normal PPC command. This allows us to detect the case where a sequence is interrupted and after the interrupt an address reference arrives first. Since sequences beginning with an address are illegal, the reference can be discarded and the sequence noted to be invalid.

- PPCs are required to maintain the sequence of the command references it does use. This allows us to check the sequence when it is not interrupted. It also allows us to use conventional techniques for checking the command references. Address references are ignored for the purpose of sequence number checking.

- PPCs are discarded on context switches, as before.

The drawback to shadow mappings is that they require duplicate entries in the page table and TLB: one for the normal mapping, one for the shadow mapping. Lenoski [Len92] points out that shadow mappings might not need page table duplication if the TLB miss handler is modified to anticipate misses for both the normal and shadow versions of the address. The downside to that approach is it prohibits finer-grain protection of the mappings, and it requires modifications to the TLB miss handler, which might impact normal-mode performance.

Providing Translations Through a MAGIC Software TLB

There is a second option available for providing authentic translations: performing the translation in MAGIC. Providing translations on MAGIC via a software TLB has several advantages. The TLB allows us to specify protocol operations without the need to communicate authentic physical addresses to MAGIC during the protocol command sequence. For example, a message send that transfers multiple pages of data can be specified as a single virtual address and length pair instead of multiple physical page addresses. The ability to translate addresses efficiently also allows us to send MAGIC-level messages across the system using virtual addresses, which is useful in some protocols.

In this section we describe two techniques for translating addresses on MAGIC, focusing on the implementation we find the most attractive, a software TLB. This alternative has different characteristics than using shadow mappings, some of which address the limitations of that technique. At the end of this section we compare the advantages and disadvantages of the different techniques for providing translations.

In the shadow mapping approach, addresses were referenced by the processor during a PPC, emerging on the bus as physical addresses. Instead, the processor could merely pass the *virtual* address down as one of the arguments to the PPC. Since PPC arguments are data values, they emerge from the processor directly, without being translated or authenticated in any way. Recall that MAGIC does not provide support for hardware address translation, as a simplification of the design. Instead, MAGIC must translate these addresses in software.

We could implement the software translations in two distinct ways. The simplest alternative is to read the page tables directly. This approach is initially attractive because it allows access to mappings for the complete address space, without the size limitations imposed by a TLB-like approach.

However, an implementation which reads the tables directly encounters two serious drawbacks. The first is that synchronization between the kernel and MAGIC would be required for access to the tables. Though it is possible to achieve this synchronization, the code is quite complicated, and it introduces OS code into MAGIC, which we have strived to avoid.[5] Since virtual memory code is deeply rooted in the kernel, implementing correct synchronization and avoiding deadlock is a difficult undertaking. The second issue is that the PP's access to operating system data structures would require care to assure the MAGIC data cache and the processor cache are consistent. Our other protocols have been designed intentionally to prevent MAGIC from caching application data, so this would represent a significant additional source of errors.

The alternative to reading the page tables is implementing a cache of translations in MAGIC we call a *software* TLB. A software TLB mimics the same functionality provided by a hardware structure. Because this TLB is kept consistent with the page tables, it would seem to introduce the same issues as reading the page tables. In fact, the problem is greatly simplified. First, the TLB only contains a small number of entries relative to the page table, so only those entries need to be consistent. More importantly, since it is a copy of the entries, no synchronization is required to access the TLB itself.

Careful design can achieve good performance using this approach: mappings are stored in a concise hash table structure that makes translation efficient and permits full associativity (though the TLB is searched serially, unlike a hardware approach). To gauge the software overhead, we hand coded and scheduled a basic but fully functional software TLB using both virtual address and process/address space matching for hit detection, as well as a write protect bit. Our implementation incurs 8 MAGIC cycles constant overhead plus 7 cycles for each entry searched. This provides excellent performance if hashing is effective: a hit on the initial probe is thus only 15 cycles (150 ns), while a scan of 16 entries would take 120 cycles (1.2 μs).

Of course, when a handler consults the TLB it may find a translation is not present. When a TLB miss occurs, the PP requests the translation from the processor with an OSPC. This OSPC can be optimized specially to make it efficient, but since it interrupts the processor the cost is still large. After the interrupt, the software overhead on the main processor depends on the kernel in use and on the status of the page itself; the common case is likely to be at least several microseconds. After performing the translation, the processor installs the mapping in the software TLB using a PPC.

Since software TLB misses are expensive, the straightforward demand-driven approach to filling the TLB may be impractical. Instead we are motivated to avoid these misses altogether by *preloading* the TLB (i.e., loading translations before they are needed).

[5]Though some dependencies on the operating system are inevitable, the vast majority of the code that implements system "policies" executes on the main processor only. The OS-level interactions with MAGIC are restricted in nature, and in most cases are required to know very little about the OS implementation and policy details.

Pre-loading the Software TLB

One approach is to have the kernel notify MAGIC when it creates new page mappings, under the assumption that these mappings may be used soon. This is unlikely to be effective, however, since the locality between mapping creations and the use of these addresses is probably too low, especially given a relatively small MAGIC TLB. A more effective technique is to *mirror* the processor TLB in MAGIC, so that any addresses currently usable by the processor would also hit in the software TLB. This would require extending the processor TLB miss handlers to notify MAGIC of the translation it installs. While this is very easy to implement, and likely to be effective in largely eliminating MAGIC software TLB misses, this technique would increase the cost of TLB misses on the processor. Unfortunately, since TLB miss handling costs can amount to a significant overhead in many workloads, increasing this overhead is likely to decrease overall performance.

We could also pre-load the TLB through explicit hints from the application. In advance of using a translation, the application could announce its intention to use MAGIC services and the translation can be installed in MAGIC's TLB. Paradoxically, since authentication is required, this must be done in a protected fashion using one of the reliable techniques (reiterated below).

1. A "null" PPC could be issued in advance, that merely touches the addresses in question. If the addresses are found to miss in the software TLB, MAGIC requests translations through an OSPC.

2. A system call could also be used, asking the kernel to issue a kernel-level PPC to provide an authentic mapping.

3. The shadow mapping technique could even be used to provide an authentic translation from user level.

It may seem strange to suggest using these techniques in conjunction with a software TLB, since one of the benefits of the TLB is to *avoid* these techniques! The motivation arises from the possibility of *reusing* translations in the software TLB, and the observation that TLB misses in MAGIC must interrupt the processor, which is much more expensive than installing the mapping in advance. Naturally, the effectiveness of reusing translations depends on the reference stream of the program, as well as the impact of effects such as multiprogramming on the contents of the software TLB. Fortunately, the software TLB may be larger than a hardware TLB structure, increasing the chances of reuse.

4.2.2 Guaranteeing Translations Remain Valid

Once MAGIC receives authentic translations through one of the techniques described above, the remaining problem is to guarantee the validity of an address throughout its use by a protocol operation.

The simplest solution is to lock the appropriate virtual pages to prevent the operating system from changing the mapping. Many systems have used page locking to protect translations, such as the Intel Paragon and Cray T3D [Int91, Cra93]. In some environments, locking pages may be a sensible way to protect translations. In FLASH, given our goal of providing a flexible, widely usable system, page locking is undesirable. There are two styles of page locking, each with its own disadvantages:

First, pages may be locked in memory for the duration of the program. This solution defeats the flexibility of demand paging, reducing the ability to share the machine. Alternatively, the pages may be locked for the duration of a protocol operation. This is common in systems where address changes are disallowed for the duration of DMA access. This latter approach is also undesirable because it requires system calls every time a user program invokes a protocol operation. This overhead can cause the primitive to be prohibitively expensive, such as the DMA hardware in the Cray T3D which required a system call. The follow-on T3E eliminated DMA support altogether, partly due to this high overhead [Sco96].

We propose three alternatives to page locking for maintaining the validity of physical addresses in use by MAGIC. The first two techniques assume that MAGIC receives authentic physical addresses as part of the user-level command sequence that describes a protocol operation. The third technique removes this requirement by supporting a software TLB that allows MAGIC to perform translations itself.

All of these techniques rely upon the operating system's mechanism for keeping TLBs consistent, commonly known as TLB shootdown. Black et al. [BRG$^+$89] describe this methodology to prevent TLBs from accessing translations rendered obsolete by page mapping changes. In TLB shootdown, a processor desiring to make mapping changes to pages visible to other processors must first guarantee that obsolete copies of the mapping have been eliminated. Eliminating TLB entries usually requires an interrupt to the involved processors since, unlike cache coherency, hardware features for maintaining TLB consistency are usually not provided at the pins of the processor. The initiator cannot make the mapping change until it knows that all the processors have removed the old mapping.

When the processor receives a request to invalidate a TLB entry as part of a shootdown, one of the things it must do is wait for all pending memory operations to complete. By doing so, it protects these operations (which traverse the system with physical addresses) from any effect of the mapping change. The techniques we describe here extend TLB shootdown to also contact MAGIC to notify it of the change. The three approaches we describe below use that notification in different ways.

The Hold-Off Technique

The *hold-off technique* extends TLB shootdown by treating alternate protocol operations as one of the operations that must complete before the shootdown can be acknowledged. In essence, this treats the operation as a memory reference. Preventing translation changes from occurring through hold-off can potentially lead to deadlock for certain types of operations. For this reason, the hold-off technique is only applicable to operations that are guaranteed to complete *on their own* without requiring any interaction with the main processors, which may require paging.

Even though the hold-off technique does not apply to all types of protocol operations, it is an extremely simple and efficient mechanism for protecting translations in many protocols. Example operations that can safely use hold-off include remote Fetch-and-Op or simple memory copy. However, operations such as a traditional synchronous message send can not use the hold-off technique since they require interactions with other processes before completion.

Hold-off is implemented using a software counter maintained by MAGIC that represents the number of outstanding protocol operations currently desiring to protect their translations. A protocol operation enables hold-off by incrementing the count, then later when the operation completes it releases its use of hold-off by decrementing it. To cooperate with hold-off, the operating system is required to notify MAGIC and to wait for a response before proceeding with a translation change. If the hold-off count is zero, MAGIC responds immediately. However, if the count is non-zero, MAGIC delays the response to the processor until the count reaches zero.

Once the processor tries to change a translation and is forced to wait because hold-off is active, we prevent the initiation of further operations that require hold-off (Operations in the middle of their command sequence protocol are conservatively forced to retry when they attempt to commit). Since the processor is prevented from initiating new protocol operations after a translation change arrives, the counter is guaranteed to return to zero as long as the previous operations eventually complete.

The Invalidation Technique

Unlike hold-off, which temporarily prevents translation changes from taking place, the *invalidation technique* merely uses the shootdown to notify MAGIC of the translation change. In response, MAGIC *invalidates* physical addresses rendered obsolete by the change. When the physical address is next used by MAGIC, the software handler detects the invalid address and requests a mapping for the page from the main processor. Below, we briefly describe the required support for this technique.

To support the invalidation technique, MAGIC must be provided with the *virtual address* as well as the physical address. If the software TLB technique is used, the virtual address is readily available. If a shadow mapping is used, we can utilize the otherwise unused data value passed to MAGIC on a shadow mapping write. Specifically, the processor writes the virtual address as the data value for the address reference. If `a` is a pointer to the memory in question, and `SHADOW(a)` represents a pointer to the shadow mapping for a, the application would make its shadow mapping reference as `*(SHADOW(a)) = a`. Though the data written is not authentic, incorrect virtual

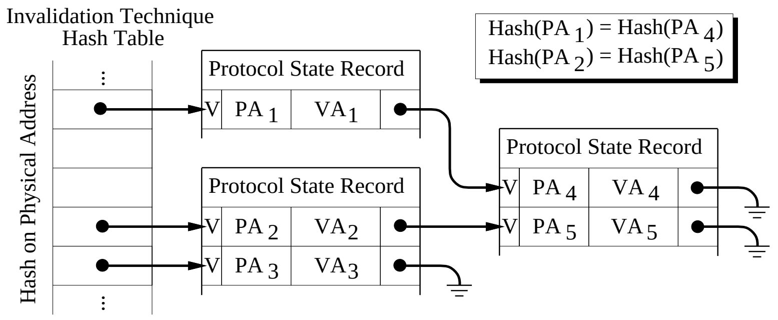


Figure 4.3: Example translation invalidation data structure.

addresses can at worst corrupt memory belonging to the requesting process and do not compromise protection for other processes.[6]

MAGIC keeps track of the physical addresses currently in use, so that it can quickly respond to shootdown requests. Physical addresses provided to MAGIC are stored in the protocol's state record and also linked into a hash table of in-use physical pages, as shown in Figure 4.3. Using this data structure, the PP can efficiently invalidate uses of a physical translation on a translation change notification from the local compute processor. The PP writes a zero into each instance of the now-obsolete physical address, since zero can be detected by subsequent handlers with a single branch. An entry in the hash table is removed when the corresponding operation completes (we use doubly linked lists so this can be done efficiently).

By convention, handlers for operations that use the invalidation technique check the validity of a physical address in each handler that uses the address. If an invalid physical page address is detected, MAGIC generates an OSPC, just as in the software TLB. This interrupts the processor and communicates the corresponding virtual page address and the process id. MAGIC resumes the operation once the processor responds with the new translation, running other handlers in the interim.

The invalidation technique is quite efficient in the common case, i.e., when translations do not change. Overhead is incurred at the end of initiation when the provided page addresses are added to the appropriate linked lists. Similar overhead occurs to remove the addresses from the list when the operation completes. The hand coded PP instruction sequence to add a translation to a list only takes 8–9 cycles to execute, assuming an elementary XOR hash function. The time required to check the address on each use is minimal, adding only a single branch instruction to the handler. A potentially larger component is checking for and invalidating obsolete physical addresses when

[6]If the virtual address is incorrect, then one of two things occurs: Either the subsequent retranslation request fails because the application does not have the necessary rights to access the region of memory, or a new translation is returned even though it does not match the same location that was originally specified. Neither case compromises the protection of *other* processes' memory.

translations change. The exact latency of this operation depends on the effectiveness of the hash table and length of the lists, which is related to the number of operations outstanding.

One of the limitations of the invalidation technique is that it can not protect remote physical addresses. For example, consider a memory copy operation from a local to a remote buffer that is implemented by sending destination physical addresses along with the data. Since the invalidation technique allows translation changes to occur immediately, destination physical addresses that are in transit could become stale. Note that this example would work correctly with the hold-off technique since translation changes are delayed until the memory copy operation completes.

Software TLB

If we use a software TLB, it is essential that it be kept coherent with the processor page tables, otherwise the translations it provides may be incorrect. Fortunately, the software TLB is able to utilize the shootdown notification from the processor directly, removing the now-obsolete mapping from its TLB if it is present. This assures the mappings in the software TLB are correct; to protect translations already in use by MAGIC, handlers must check the TLB on each use of an address, to be sure the translation is still valid.

We can reduce this latter source of overhead by using a software TLB in conjunction with the invalidation technique. By using the invalidation technique normally, the PP is able to keep track of outstanding physical addresses, and can invalidate them if the translation changes. This avoids the need to retranslate in the software TLB on each handler, but instead the simpler test against zero is sufficient. Alternatively, a variant of the invalidation technique can be used. Instead of keeping track of where each use of a physical address is stored, protocols can keep a pointer to the TLB entry where the translation was found and quickly check that the address is still present. This variant reduces the cost of maintaining data structures for the invalidation technique and changing translations, while increasing the cost to verify an address is valid on each use.

4.2.3 Comparing the Virtual Memory Techniques

In this section, we have described a family of techniques for both providing translations authentically and then protecting them while in use. We briefly review these techniques to illustrate some of the trade offs between them.

For *providing* translations, the shadow mapping technique is advantageous for its low overhead and efficient utilization of existing processor translation capability. We use the shadow mapping technique for providing translations in the memory copy protocol described in Chapter 5. However, this technique does have several disadvantages. The most obvious is that it requires extra address mappings to be exported to the application. These may consume additional TLB entries on the main processor that may increase the number of TLB misses, reducing performance. A software

TLB addresses avoids this problem because now only one processor TLB entry is needed—the one that maps the desired PPC area. Instead, virtual addresses are passed to MAGIC and translated there. The drawback to the software TLB approach is that its performance is closely tied to translation caching effectiveness. Furthermore, misses in the MAGIC software TLB are very expensive, since they require processor interrupts, so pre-loading the TLB somehow may be required to achieve good performance from this technique

For *protecting* translations, we described three techniques that trade off efficiency for functionality. While hold-off is by far the most efficient technique, deadlock can arise if hold-off is used in some scenarios, so it must be avoided for those protocols. On the other hand, certain circumstances *require* the use of hold-off to protect physical addresses in flight in the machine. The memory copy protocol we describe in Chapter 5 must use hold-off for that reason. The invalidation and software TLB techniques are somewhat more general and can avoid the deadlock cases of hold-off, but each entails higher overheads. Fortunately, the flexibility of MAGIC allows us to use the most appropriate technique for each type of protocol operation, in addition to experimenting with some of the hybrid approaches we described to improve performance.

In a recent paper, Schoinas and Hill study the issues for address translation in network interfaces using a range of approaches [SH98], many similar to the those presented here. Their paper refers only to our software TLB technique but fails to refer to our other techniques for providing and protecting translations [HGDG94, HBG$^+$97]. They present simulation results comparing their approaches, ultimately concluding that software-based techniques similar to ours are usually sufficient, though the ability to perform translations (i.e. read page tables) in the NI is ideal if tractable.

4.3 Protection

The techniques described above allow a user process to specify protocol operations that are executed by MAGIC on the process' behalf. Our goal in FLASH is to provide protection for the services provided by MAGIC, just as we would for any other system services provided to the user. We consider briefly the kinds of protection that are needed, while noting that many of the protection features are protocol-specific and are described in later chapters.

The most critical feature is assuring memory protection in MAGIC operations—the same level of protection the MMU would provide if the processor carried out the operations itself. To a large extent this is achieved as part of the translation process described in Section 4.2. The shadow mapping approach leverages the processor MMU directly, so protection is assured. Assuring protection for translations generated by a software TLB is slightly more complicated. We must take care to inform MAGIC of the process (i.e., protection domain) which invokes a MAGIC operation so that software TLB misses can be satisfied in the address space of that process. As mentioned earlier, the operating system notifies MAGIC about the PID of the locally running process at context switch points, so this information is readily available.

Other protection features are provided to address the low level semantics of the operations we provide. For example, range checks must be made on arguments provided to MAGIC to prevent applications from accessing state which is invalid or belongs to other applications. When interaction with other processes is required (e.g., the interrupts which can be caused by the delivery of active messages), the protocol should assure that a process only sends to receivers interested in communicating with it.

In the subsequent sections, we consider the details of protection relevant to the particular protocols we implement. In certain cases, such as providing memory copy-based block transfer, memory protection is sufficient. In others, additional protection features are needed.

4.4 Coherence of Alternate Protocols (PP Ownership)

The purpose of most of the alternate protocols we describe is to manipulate system memory in some useful way. MAGIC is particularly appropriate for many data manipulation tasks since it has an efficient memory access path based on its central location in the node. However, accessing memory is not sufficient, MAGIC may need to get the current data from a processor cache. Similarly, if it modifies memory, it must also keep processor caches coherent if its alternate protocols are to be widely usable.

Clearly, FLASH provides the ability to keep caches coherent through the flexibility of MAGIC. The challenge we face is to integrate coherent memory manipulation with our alternate protocols while achieving good performance and correct behavior. Unfortunately, the full cache coherence protocol is complicated and very large.

Furthermore, integrating cache coherence with alternate protocols is not as straightforward as it might seem. Intuitively, one might imagine that many cache coherence handlers could be leveraged for alternate protocols directly. In nearly every case, these handlers take *almost* the needed action, but not *exactly* the right one. We could make the handlers more general by installing checks to allow them to perform one kind of coherence operation or another. Though this fosters direct code sharing, it slows the handlers down in every case, including the case where cache coherence is used by itself. We did not consider slowing down the common case for cache coherence to be an acceptable option.

Instead, we address two alternatives for maintaining coherence for alternate protocols, *custom coherence support* and *PP ownership*. The choice between these options depends on the amount of data processing which an alternate protocol provides. We explore protocols in later chapters that use each of these techniques.

Custom coherence support is, as the name implies, a specialized implementation of coherence handlers for a particular alternate protocol. In a custom implementation, a set of handlers is required that contains all the functionality needed to extract data from processor caches, update and maintain the directory structure, and maintain coherence as needed (such as by sending invalidations, etc.). The nature of a custom protocol like this is that it is used for a particular alternate protocol only. In Section 5 we describe in detail the custom coherence protocol we implement to provide memory copy in FLASH.

In this section, we consider the other alternative, which we call *PP ownership*. In many protocols, the requirements for coherently manipulating user data may be very mild. In the case of Fetch-And-Op, for example, a single data word is manipulated, but it must be done coherently. These situations do not warrant the complexity and code overhead of a custom coherence protocol. Our approach in these cases is to leverage the *existing* cache coherence protocol to allow the PP, like any other processor, to participate in the coherence protocol and take ownership of a line to modify it coherently. Thus the advantage of PP ownership is that it can be used from any protocol without duplicating the coherence support. The cost of this reduced code size is that PP ownership is somewhat less efficient, so it is only useful in protocols with mild requirements, as described above.

To allow the PP to take ownership of lines, we leverage extensions that the cache coherence protocol provides for the PCI IO subsystem. We briefly describe this support, then explain how it can be modified to allow the PP to take ownership of lines.

4.4.1 MAGIC IO Subsystem

As described in Section 2, MAGIC contains a PCI IO interface, which is one of the three hardware interfaces that can make requests of MAGIC. Unlike the processor, which maintains and controls a large cache, the PCI IO bus has much milder requirements. To provide coherent IO operations, MAGIC allows the IO system to take ownership of a *single line* at a time. Since the bus has no cache in which to store the line, it is instead held in a data buffer which is “loaned” to the IO interface while in use. This allows the IO system to modify pieces of the line coherently — necessary since PCI only manipulates 32 bits of data at a time.

Since IO can take ownership of lines, the cache coherence protocol is required to provide several functions which go beyond the basic coherence protocol:

Request a line for the IO system Unlike normal coherence requests for data, which originate in a processor, this special request indicates that the data should be sent to the IO system on the requesting node. FLASH requires the data to be explicitly sent to the IO unit, so this message type is required to distinguish processor and IO requests.

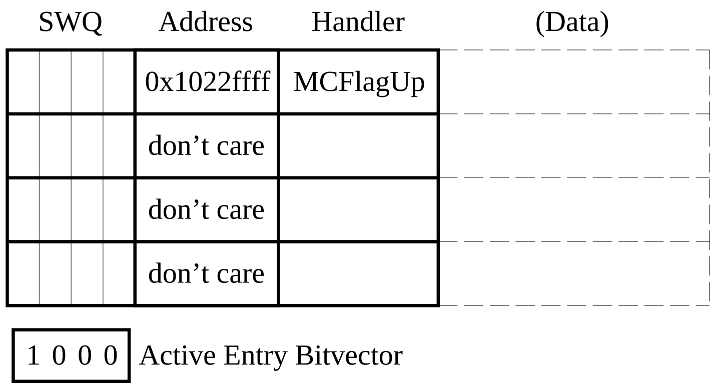


Figure 4.4: Example PP ownership table, which stores the currently outstanding requests.

Request a line currently held in the IO system Since the IO system may be the current owner of a line, the coherence protocol must track the line's presence in the IO system and may need to request it back to satisfy a processor request

4.4.2 PP Ownership

PP ownership leverages the protocol extensions for IO to allow the PP to act as if it were part of the IO system. Building upon these protocol features—which are already present—allows alternate protocols running on the PP to manipulate data coherently with only minor modifications to a few cases in the coherence protocol.

First we establish a data structure that holds outstanding PP ownership requests, illustrated in Figure 4.4. When the PP wants to request ownership it selects an available entry and fills in the memory address and the address of a handler to execute when ownership is acquired. The request may also store optional data arguments to be passed to the handler when executed. We implement several support routines which read this data structure and send the request. The advantage of a central implementation of these routines is that all alternate protocols can share PP ownership's code directly and avoid any code duplication. For reliability, the PP ownership table contains a software queue record to allow the handlers to retry requests if needed.

The support routines send a request to the home of the memory line, just as the IO system would. Eventually the line arrives at the node, flagged to be delivered to the IO subsystem. Normally the cache coherence protocol would blindly send this to the IO unit; we modify the reply handler to first check the table of PP ownership requests. If it finds a match, it invokes the registered handler instead, thus providing the PP with ownership and the most current copy of the data in a data buffer. Once the PP is finished with the line, it calls a different routine to cleanly write this data back to the home, just as it would from a processor or IO system writeback.

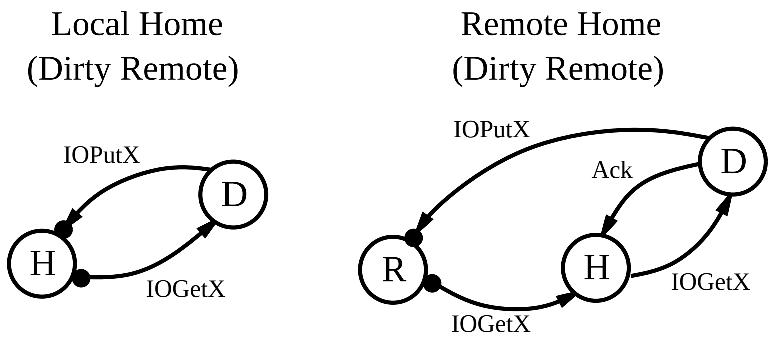


Figure 4.5: Example cases of PP Ownership building on top of the I/O support in the protocol. Solid black dots indicate the only places where the protocol can tell the request is actually PP ownership and not IO. Special actions are only required in those situations.

Figure 4.5 illustrates how this works, both for local and remote addresses (in the remote home case, the node labeled "R" is the requester). This figure shows the case where the desired line is dirty in a remote cache. The solid black dots indicate the only locations where special handling is required to support PP ownership. By isolating the fact that the request is actually PP ownership, the other handlers in the protocol can be used without modification.

Though we have modified the cache coherence protocol slightly to provide PP ownership, the performance impact of the change should be negligible in the vast majority of cases. First, only the IO extensions to the protocol are modified. Handlers for normal processor cache coherence are not affected. In the IO handlers when PP ownership is not active, checking the table is very efficient and the request can quickly be sent on to the IO unit. When PP ownership is in use, actual IO operations incur mild overhead to check the table and therefore take longer. Fortunately, since the IO system is generally used by low-performance devices such as disk and Ethernet, the performance of these devices is unlikely to be affected by a delay of even hundreds of MAGIC cycles (i.e., 1–2 microseconds).

The drawback to PP ownership, as we have alluded, is its overhead. This arises from the need to guarantee acquisition of ownership despite temporary problems and corner cases that may arise. To provide this assurance, we use the software queue and the other tables, all of which take time to set up. Fortunately, in some cases, such as when the requested line is local and the directory indicates the data is readily available, we can bypass some of these checks. In this "fast path" scenario, the overhead of acquiring local PP ownership is 68 MAGIC cycles (0.68 μs) and releasing it is 48 cycles (0.48 μs). If the fast path does not apply and the fully robust version is required (e.g., for remote memory lines) the overheads increase to 169 cycles (1.69 μs) on the requesting node to acquire and 40 cycles (0.40 μs) to release.[7]

[7]In the case of remote lines, the numbers listed are the *overheads* introduced by PP ownership. Not included is the time at the home to request the line, that varies depending on its caching state as usual.

PP Ownership Summary

PP ownership provides a mechanism for the occasional coherent access of data by the PP. Its goal is to leverage existing handlers to the fullest extent possible, and to allow this support to be shared by any alternate protocol. Despite its benefits, PP ownership is not well-suited for accessing large amounts of data. In the next chapter, we describe how a custom coherence protocol can provide significantly higher performance for a data-intensive memory copy protocol. In that protocol, even though we provide custom coherence support, we still use PP ownership in a few situations, to reduce the size of the protocol in corner cases where performance is less critical.

4.5 Summary

This chapter described a number of software mechanisms that are useful for the alternate protocols we study. We consider these mechanisms separately since they are features that the various protocols we study have in common.

Several of the mechanisms address the interaction between the processor and MAGIC. We described several techniques by which the processor can make requests of MAGIC, for example to initiate alternate protocol operations. Similarly, we showed how MAGIC can in turn make requests of the processor, e.g., to request a page be mapped. We also described extensions to this interface to support virtual memory with protection, allowing it to be used in the context of a modern operating system.

We also introduced the difficulty in achieving cache coherence for alternate protocols within MAGIC. Custom coherence protocols are used in some cases for peak performance. In this chapter we described a technique called PP ownership that is more appropriate for protocols needing only occasional coherent updates.

Chapter 5

FLASH Memory Copy

In this chapter we consider in detail the design, implementation, and performance of the FLASH memory copy protocol introduced in Section 3.1.4. The goal of this protocol is to provide high performance data movement supported by MAGIC and exported directly to the user through a memory copy interface. Furthermore, our ultimate aim is to fully integrate this protocol with cache-coherent shared memory. This allows shared memory programs to use the protocol transparently to accelerate block data transfer, in addition to more traditional uses for implementing message passing. By leveraging the processor/MAGIC communication features described in Section 4.1, we export this facility to the user without system calls but while maintaining protection and the ability to interact with virtual memory.

We begin in Section 5.1 by describing the application programming interface to this memory copy facility. Section 5.2 details how a restricted version of this interface with only limited coherence support can be implemented using the embedded programmability of FLASH. In Section 5.3 we discuss the challenges and benefits of providing a fully general implementation of that interface, then describe extensions to the basic implementation it requires. We study the performance of the protocol in detail in Sections 5.4–5.5, characterizing its behavior using a variety of benchmarks and applications. We conclude in Section 5.6 by discussing related work in this area.

5.1 FLASH Memory Copy Application Programming Interface

A fast block transfer primitive is important for both applications that exhibit large data movement and the operating system itself. In addition, it can serve as a building block for message passing protocols such as the Message Passing Interface (MPI) standard. In fact, in response to our previously published work and to that of others, recently at least one new lightweight messaging protocol has to emerged which effectively utilizes the kind of high performance communication we advocate [BSSD96].

Unlike traditional message passing machines in which message passing is the only communication model, FLASH also provides cache-coherent shared memory functionality to the user. By integrating the block transfer protocol with cache-coherent shared memory, we enable applications to seamlessly utilize block transfer in shared memory programs as well. Frank and Vernon advocate a similar approach with slightly richer semantics that integrates the ability to perform explicit transfers as an optimization within a shared memory environment [FV93].

Our application programming interface for block data transfer is an version of the well-known C-library bcopy call:

```
void fbcopy (char* src, char* dest, int len, uint64* flag);
```

Fbcopy copies the memory specified by the `src` parameter (called the *source buffer*) to the address specified by the `dest` parameter (called the *destination buffer*). `len` specifies the number of bytes to copy. We extend the normal bcopy interface slightly by providing an additional parameter that points to a user-visible completion `flag` incremented by MAGIC when the transfer has finished. Since fbcopy proceeds asynchronously to the application, the flag allows the main processor to perform other calculations in parallel, if desired, then later test the flag for completion.

Given our goal of providing a single facility to be useful in a wide range of situations, we export block transfer to the user in this straightforward way rather than constrain the user to a more rigid interface. This allows programs to build higher level abstractions on top of this facility very easily, rather than conform to a very specific model which may have undesired characteristics. This issue is analogous to the debate between CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) architectures. Some CISC architectures provided instructions that improved performance but were difficult for the compiler to identify or so specific they were not widely applicable. Our choice of a simple primitive takes a distinctly RISC-like position, providing instead a fundamental operation which can be used to build up more complicated semantics.

5.1.1 Sources of Complexity

Though the programming interface for the block transfer primitive is straightforward, an efficient implementation of this interface on MAGIC is nontrivial. While FLASH does provide the features we need to implement block transfer, in practice we find that only through careful design of the data transfer handlers can we achieve the levels of performance we desire.

Moreover, two issues arising from the integration of block transfer with shared memory serve to increase the difficulty of this implementation. Figure 5.1 illustrates these sources of complexity.

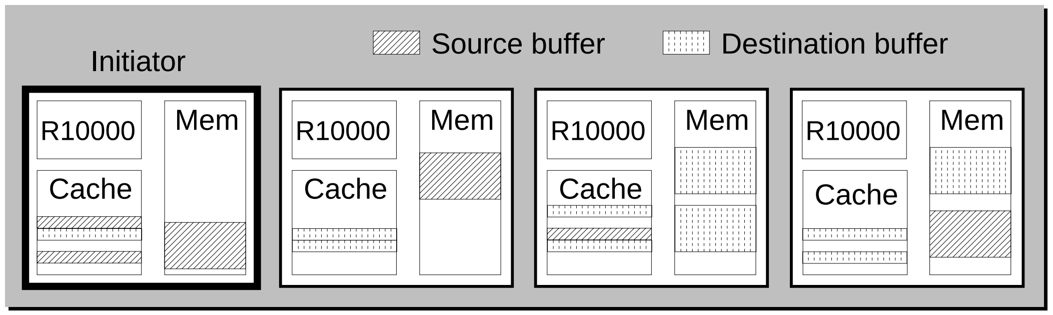


Figure 5.1: Sources of complexity in memory copy. Buffers may be cached in any node on the system, and physical pages of the block may be allocated out of remote memory.

First, the cache-coherent shared memory support provided by FLASH allows the source and destination buffers to be cached in any node of the system. For the source buffer this means that the current value of the data might be in another node's cache; when the transfer is carried out the protocol must retrieve the data from that node. Similarly, nodes caching the destination buffer must be notified when the block transfer modifies the buffer so they can invalidate their old copies.

The second issue is a related one at the page level. The source and the destination buffers may span multiple virtual memory pages, resulting in non-contiguous physical pages. Furthermore, the physical pages may be scattered across different nodes of the machine for many reasons: process and page migration for load balancing, sharing of data among cooperating processes running on different nodes, and operating systems which can allocate pages on remote nodes [CHRG95]. As a result, it is quite possible for the node requesting the transfer, called the *initiator*, to be distinct from the home nodes of the source buffer pages. The added complexity for handling this case arises because coherent operations on a line must consult its directory state (located at its home node). We refer to this issue as *remote home support*.

Coherence Models

The first source of complexity described above—the choice of the coherence model for message data—is a key issue in implementing data transfer on cache-coherent shared memory systems. Below we discuss the various options and their corresponding tradeoffs. Kubiatowicz et al. present a similar categorization in the context of message passing in Alewife [KA93].

The simplest option is to provide *no coherence* for block transfer data. This corresponds to reading the data directly from the source buffer and storing the data directly into the destination memory without taking any coherence actions. This option is not acceptable in practice since it precludes caching of either of these buffers. The second option, called *local coherence*, provides coherence if the message data is uncached or cached only at the home node. This closely matches the functionality provided by most message passing architectures where each processor can only cache the data that resides in its local memory. The third and most general option is to provide *full integration with coherence* (often referred to as *global coherence*) which imposes no restrictions on the caching of data. The additional functionality provided by fully integrating with coherence is essential to make block transfer widely usable (e.g., memory copy used to achieve page migration in a cache-coherent system). In cases when the lines are clean in local memory or are cached only locally, full integration with coherence does essentially the same work as local coherence. Extra coherence transactions are required only when message lines are cached remotely or have remote homes.

We begin below by showing the implementation details for a simplified version of memory copy that supports only local coherence. By using a slightly simplified model, we can focus on the details of how transfers are fundamentally accomplished. In Section 5.3, we explain how this base implementation can be extended to achieve full integration with coherence.

5.2 Locally Coherent Transfer Model

In this section, we describe a version of block transfer that provides only local coherence support. This model is very similar to that provided in message passing machines where data cannot be cached remotely and only local memory can be referenced by the processor. This restricted model is used to show how we accomplish transfers in FLASH. The specific restrictions it entails are:

- The source buffer is only allowed to be cached by the source node, and the destination buffer is only allowed to be cached at the destination node.
- The transfer can only be initiated by the processor on the home node of the source buffer.

5.2.1 Transfer Overview

The application initiates a block transfer by invoking the fbcopy library call (defined in Section 5.1), which describes the transfer to MAGIC using the memory mapped interface described in Section 4.1. An fbcopy call transferring one page requires four uncached writes to MAGIC. These writes contain the length of the transfer and the physical addresses of the source buffer, destination buffer, and completion flag. Longer transfers may require more writes for the additional pages involved.[1] MAGIC maintains a pool of state records used to store the description of the transfer during initiation. The same record is used to maintain the intermediate state that encapsulates the current status during the transfer. Physical addresses stored within MAGIC memory copy state records are protected during the transfer by the hold-off technique described in Section 4.2.2.

[1]When using the double mapping technique described in Section 4.2.1, each page must be referenced as part of the initiation, since they may not be physically contiguous. If translations are provided by a Software TLB, on the other hand, the processor can specify a virtual address and a length, potentially reducing the number of commands in the initiation sequence.

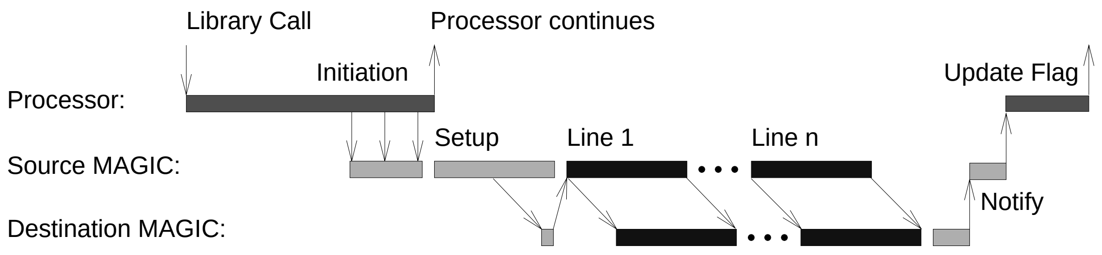


Figure 5.2: Schematic transfer timeline, illustrating the processor and the source and destination MAGIC chips. Medium gray bars are for transfer initiation and flag update, light gray bars are for transfer setup and completion detection, and black bars are for the transfer and receive handlers.

Once a transfer is accepted by MAGIC, the *transfer handler* at the sending node sends the data as a series of cache lines. For each line, the transfer handler (running on the protocol processor) reads the appropriate data from memory into a data buffer in MAGIC, adds header information to form a FLASH message, and sends it to the destination node. We label the line at the source node with its destination address so that it can be written to memory quickly. When it arrives at the destination, it runs a handler to process the message, called the *receive handler*. In this case, the receive handler stores the accompanying data into the destination memory as part of the transfer. The overall transfer process is illustrated in Figure 5.2.

We use the software queue to carry out the block transfer through multiple invocations of the transfer handler at the sending node. Each time the message transfer handler is invoked, it sends one or more lines of the message data, updates the transfer state, and reschedules itself using the software queue. Note that even though each line of data is sent separately, the handler has the option of sending multiple lines during each invocation. This technique, referred to as *chunking*, has the potential to improve performance by amortizing the overhead of starting the transfer handler and saving and restoring the transfer state. We evaluate the effect of this optimization in Section 5.2.3.

Transferring data as a series of cache line sized messages instead of a single large message has several advantages:

Deadlock avoidance. FLASH is able to avoid deadlock using a combination of software conventions and Inbox hardware assistance. As described in Section 2.3.6, to avoid deadlock MAGIC must stop sending further messages and yield the PP if outgoing queues fill. If block transfers traveled as one large message, the FLASH conventions would be insufficient to avoid deadlock and would have to be completely redesigned. Deadlock is a sufficiently critical and difficult issue that leveraging the existing deadlock techniques is invaluable.

Coherence of transfer data. The coherence protocol in FLASH maintains directory information at cache line granularity. As we describe in Section 5.3, the complexity of maintaining coherence of message data is reduced, and thus performance can be increased, when the transfer size matches the coherence granularity.

Network and fairness advantages. The memory system and MAGIC are designed and optimized for transferring cache lines and therefore cache line packets are handled efficiently. In addition, allowing the service of cache-coherent operations to be finely interleaved with large transfers is beneficial for achieving efficient overlap of computation and communication. The network may also perform better if large messages are broken into pieces instead of traveling as a single huge chunk.

Of course, the disadvantage of sending lines individually is that each one incurs two sources of overhead: network headers and protocol processing. First, the message headers each line carries in the network decrease the useful bandwidth. If we sent a larger block, headers would account for less of the bandwidth used. Second, when the lines arrives at the destination, each runs a handler to process its contents, as opposed to a handler processing a larger block. Fortunately, the impact of these overheads is mitigated somewhat in FLASH due to the large cache line size (128 bytes). FLASH message headers add 16 bytes of overhead (11%); the handler overhead is studied later in this chapter.

At any one time, the transfer concerns itself only with the data on the current source page. That is, the source address never crosses a page boundary during an invocation of the transfer handler. Instead, we wait for the data on the page to be received at the destination and acknowledged before we advance the transfer to consider the next page. Breaking up multi-page transfers into single-page ones reduces the complexity and overhead of transfer bookkeeping and state maintenance. In Sections 5.2.2 and 5.3.2 we describe some of the sources of complexity that make this a good engineering tradeoff. While breaking up multi-page transfers reduces complexity, it still allows a substantial number of lines to be in flight (32 lines/4 KB page). Allowing many lines to be outstanding is important for hiding message latency, especially when maintaining coherence [WGH+97].

Detecting Completion

An important problem to solve in the block transfer protocol is determining when all the data has been committed to the destination memory. Only at that point can the destination buffer be made available to its consumer; similarly only then are modifications to the source buffer guaranteed not to affect the transfer. This problem is made even more difficult because we transfer data as a series of cache lines instead of a single large block. For example, some lines of the transfer may fail or be refused due to races or resource limitations. This requires the destination node to maintain a count of remaining message data so that an acknowledgment can be sent to the source when the final line arrives.

To facilitate this, the source node announces a transfer to the destination node before it begins. This message is often referred to as an *envelope*, since it contains the “wrapping” information regarding the transfer. The envelope contains two pieces of information: the source’s message ID, which uniquely identifies the transfer at the source node, and the message length. When the destination receives the envelope, it allocates a corresponding receive record for the transfer.[2] Since we count lines, in addition to its destination address each line must indicate which transfer it belongs to. One approach uses the source message ID to indicate the transfer to which it belongs. This ID is used to search a hash table to find the corresponding receive record. The remaining message count in that record is decremented, and if zero then the transfer is complete. The destination notifies the source with a transfer acknowledgment, labeled with the source message ID.

Though initial versions of the protocol followed the hash table approach, we quickly realized that the hash table lookup was a significant source of overhead. A superior approach is to reply to the envelope with an acknowledgment that provides the *destination’s* message ID, which uniquely identifies the *receive record* that was allocated. The source can then label each line with the destination message ID, which indicates the receive record number directly.

When the transfer is complete and the source receives an acknowledgement from the destination, it updates the application’s completion flag using the PP ownership technique described in Section 4.4. Figure 5.2 illustrates when the flag update occurs. We employ PP ownership to avoid implementing custom coherence support merely for the flag update. The coherence support in fbcopy in Sections 5.2.2 and 5.2.2 is designed for the handling of bulk data and is not suited for use in updating the completion flag.

5.2.2 Base Implementation

The performance-critical core of the protocol is the matching pair of handlers which send and receive the data lines of the message. We refer to these as the *transfer handler* and the *receive handler*. We begin by examining these handlers to illustrate the tasks they must perform to realize a block transfer on FLASH. In Section 5.2.3, we show how this base implementation can be improved significantly using a number of software optimizations. The ability to tune the protocol and perform extensive software optimizations to improve performance is one of the key advantages of the FLASH architecture.

[2]The envelope is merely refused if a receive record is not available, which provides flow control by limiting the number of simultaneous transfers destined for any one node. If we only relied on source-side flow control, nothing would prevent the destination from receiving many transfers simultaneously and becoming a hot spot.

Transfer Handler

The transfer handler is responsible for sending the actual data lines of the block. This entails several key functions:

- Locating the most up-to-date copy of the data. (This is important in the fully coherent model we discuss later.)
- Re-mapping the source address to its corresponding destination address. Unlike normal cache coherence operations in which a line never changes address as it moves around the system, in a memory copy the source data must be stored at the *destination address*. This re-mapping is straightforward in simple cases, but page crossings in the middle of the message and lines requiring retry complicate this task.
- Maintaining the state of the transfer as it proceeds.
- Logging lines which experience problems during the send, or which the destination node is unable to handle when they arrive. These lines must be retried so the transfer can complete.

Figure 5.3 presents the pseudo code for the base implementation of the transfer handler providing local coherence. We briefly discuss the details of this handler to highlight some important issues and illustrate the concepts clearly before delving into optimized versions in later sections.

The transfer handler is scheduled onto the software queue since it needs to execute repeatedly. As part of its scheduling task, the Inbox periodically selects the handler at the head of the software queue instead of one of the hardware queues. When the handler starts up, it ascertains the current position within the transfer by reading the memory copy state record. It immediately consults the directory state for that address, and determines the appropriate action to take.

Recall that in the local coherence model we only allow lines to be cached in the local node (if at all). We distinguish between the data being in the memory or the cache because FLASH requires the protocol to explicitly request a dirty line from the local processor. This is in contrast to snoopy bus-based designs where the same request checks both the memory and the processors' caches. Figure 5.4 shows the pseudo code for this implementation. We describe the various coherence cases below:

The line is busy (pending). The coherence protocol may mark a line as busy if coherence operations on the line are in progress. This prevents conflicting operations on a line from occurring simultaneously. In this implementation, if we find the line is busy we yield and try again later.

The line is clean in memory. In this case the data is read from memory, the message headers are prepared, and the message injected into the network. Note that since MAGIC is only *reading* the memory, outstanding shared copies require no special handling.

```
void MemCpyFullAlignedTransfer()
{
   Allocate data buffer;
   if (no buffers available) {
      Reschedule;
   }

   Read transfer state and address pointers;
   Read directory state for source line;
   if (Line busy) {
      Reschedule;
   }

   if (Line Dirty) {
      Form request for cache intervention;
      Request line from cache;
      Wait for cache controller reply;
      if (Request Failed) {
         Reschedule;
      }
      Write line to memory;
      Update directory;
      Form header for destination;
      Send message;
   } else {                    // Clean
      Read line from memory;
      Form header for destination;
      Send message;
   }

   Update remaining count;
   if (Transfer done) {
      if (Lines experienced problems) {
         Switch to cleanup mode;
         Reschedule;
      } else {
         Unschedule;
      }
   } else {
      Reschedule;
   }

   Update source and destination pointers;
   Check for destination page crossing;
}
```

Figure 5.3: Pseudo code for the base implementation of the locally coherent transfer handler.

The line is dirty in the processor cache. In this case we are required to explicitly fetch the line from the processor's cache. We request a shared copy of the line, allowing the processor to keep a shared copy for itself, rather than eliminate it altogether.[3] We must examine the result of the request for a very important reason: the processor may be writing the line back when the request is made. Since the FLASH processor interface does not consult the queue from the processor to satisfy MAGIC's request, the request may fail to find the line in the cache (indicating the writeback race). If this occurs, we are forced to yield the PP to allow the

[3]As discussed in Section 2.3.3, the R10000 does not provide a way to access the dirty line but still leave it in the dirty state, thus our request for the data steals exclusive ownership of the line at the very least.

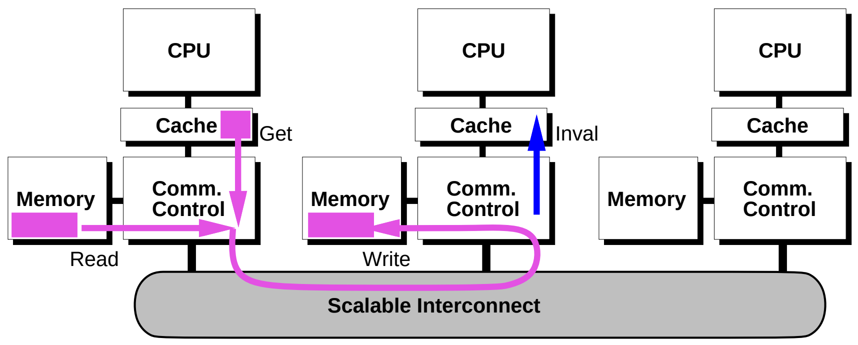


Figure 5.4: Coherence handling in the locally coherent transfer model. At the source, the line must be explicitly requested from the cache if needed. At the destination, outstanding copies must be eliminated

writeback to occur—no other action can allow MAGIC to access the most current copy of the data.

If the access indicates the dirty copy was found, the PP can proceed, preparing the message for the network. The data follows from the processor several cycles later; the data buffer logic correctly synchronizes the merging of the data with the message. The PP first writes the data to memory, then updates the directory state to indicate the line is clean in memory (with the local processor as a sharer).

After handling the coherence actions, if needed, the PP forms the destination headers, labels the line with its destination address, and issues it into the network. The remaining work in the handler updates the state of the transfer in preparation for the next line, tracking the amount of data remaining. If the transfer is complete, a check is made to see if any lines need to be retried (the retry mechanism is described below), and if so a cleanup handler is selected to run next. If no lines need to be retried, the handler removes itself from the software queue. In the normal case, the transfer merely reschedules to send subsequent lines.

Finally, the source and destination pointers are updated. Our transfer implementation requests acknowledgments from the destination node for each page, thus a source page crossing never occurs in this handler. Source page crossings are handled instead by support routines that run when the acknowledgment arrives from the destination. Since the destination address is unrelated to the source address, however, the destination address may cross the page boundary at a different point, so we must check explicitly for that condition. If the destination address crosses the page boundary, we look up the subsequent page address (supplied during initiation) since it need not be physically contiguous with the previous page.

Receive Handler

When a line arrives at its destination, our goal is to store it to memory and update the count of remaining messages as quickly as possible. Since this is an ownership-based protocol, it is normally forbidden for a processor to write a line unless it has acquired exclusive access. Fortunately, in most cases we can atomically (that is, within the same handler) reclaim ownership of the line, write the data to memory, and update the directory state to indicate the memory copy is current. We explain below how some cases prevent this operation from succeeding when the message arrives.

We previously described how we pass the destination's message ID as part of each data message. This allows the destination to find the transfer state quickly. In parallel, we also look up the directory state for the destination line. Just as in the transfer handler, we may find the line in several different states. Unlike the source side, since the destination node is *writing* the memory, outstanding shared copies must also be eliminated to maintain coherence. The coherence cases at the destination are handled like this:

The line is busy (pending). Since the directory is busy, the arriving line can not be accepted. We are forced to send a negative acknowledgment to the home.

The line is clean in memory. We write the data and leave the directory state unchanged.

The line is cached locally. We write the data, send an invalidation to the local processor, and reset the directory state.

The line is dirty in the processor cache. Normally, a cache-coherent machine only allows the exclusive owner of a line to modify it. Since our protocol writes the contents of the entire destination buffer line at its home, we can modify the protocol to improve performance by writing the line *without* acquiring ownership in advance. We then send a special flavor of invalidation we call a *dirty invalidation* to the remote node, telling its cache to discard its dirty copy of the line without writing it back.

Unfortunately, just as in the transfer handler, the processor might be in the midst of writing the line back and so our cache intervention may fail. If this occurs because our invalidation failed, it squashes the block transfer data that has been written to memory. To detect this race, the invalidation message checks the response from the cache. If the line is being written back, a negative acknowledgment message is sent back to the destination node, triggering a retry of the line from the source.

If the coherence processing for the line was successful, we update the count of lines remaining in this transfer. If we find that the transfer has finished, we acknowledge the source, indicating that the portion of the transfer announced by the envelope is safely committed to the destination memory.

Retrying Failed Lines

During the transfer, the transfer and receive handlers may encounter lines which experience problems in their delivery. For example, we described how the receive handler is forced to send a negative acknowledgment to the home in some cases. In Section 5.3 we show that full integration with coherence adds still more cases where lines may fail at the source, destination, or elsewhere in the system because some participant is temporarily unable to cooperate.

To address the range of failure cases, we maintain state in the transfer record to indicate if lines experienced problems in delivery. This is implemented as a bit vector in which one bit represents the state of each line. When a line receives a negative acknowledgment from the destination node, we calculate its offset within the transfer page and set the corresponding bit. For page sizes up to 8 KB and 128 byte cache lines, the 64 bit PP allows us to pack this vector into a single doubleword (larger page sizes would dictate a more complicated implementation).

As shown in Figure 5.3, once the transfer completes its first attempt at sending the current page, it consults the bit vector to see if any lines have received a NAK. If so, it enters *cleanup mode*. The cleanup handler locates set bits in the retry word (indicating failed lines) and regenerates those requests. When it resends a line, it clears the related bit in the retry word so that subsequent executions retry other failed lines. Retry messages are identical to the original request, and are handled exactly the same way; they themselves may fail for similar reasons.

Improving Retry Performance

Generating a retry of failed line is very similar to the initial attempt at the line. However, since random lines of the page may have failed, the performance of the handler in sending these lines is lower than the initially sequential transfer. In part, this performance penalty comes from the need to perform two address mapping functions as part of retry. The first mapping function concerns the handling of negative acknowledgments. When negative acknowledgments arrive at the source, they carry with them the destination address they were given. This address must be *reverse mapped* to generate the source address to which it corresponds. Second, when the retry is later being generated, we need to regenerate the destination address or *forward map* the source address to its corresponding destination address.

The main difference between the initial transfer and retries is that we do not have a current pointer into the destination buffer from which to address the retry message. Instead, we must explicitly calculate the destination address based on the offset into the send message. Complicating this calculation is the possibility of page crossings in the destination buffer. Figure 5.5 illustrates this complication, showing the source and destination buffers of the transfer (shaded), and the page boundaries (heavy lines) for each buffer. Since the buffer addresses are user-supplied, there is nothing to prevent them from having different page offsets, as shown. In general, they are not physically

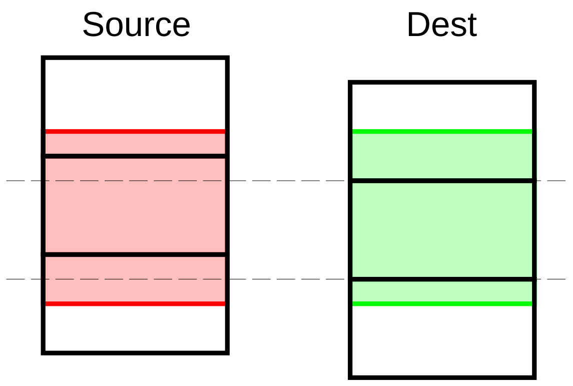


Figure 5.5: Page boundary crossings. Crossings may occur at different locations in the source and destination buffers. Page boundaries (heavy lines) are shifted to indicate the offset at which they occur in the transfer.

contiguous, so the physical address changes at page boundaries. Dashed lines show how mid-way through a source page a destination buffer page crossing requires the transfer handler to change to a new destination page.

To reduce the cost of handling destination page crossings, we pre-compute several useful constants in the initialization of the transfer. These constants, which digest the addresses of the source and destination pages and the page crossing locations, allow us to perform these re-mapping functions more quickly.

5.2.3 Optimized Implementation

The base transfer implementation described provides the core block transfer functionality we desire, but does not achieve the efficiency the underlying hardware provides. There are several software techniques we can apply to this basic implementation to significantly improve its performance. The first of these techniques targets the long latency to access lines in the processor cache. The second technique amortizes the cost of the transfer over several lines, reducing the effective overhead.

Hiding Cache Access Latency

One drawback to the base implementation is the cost of handling lines that are dirty in the local cache. Note that in the pseudo code shown in Figure 5.3, the request to the processor cache is immediately followed by spinning on the cache reply. The processor's delay in providing this response is at least 15 MAGIC cycles. During this delay, the base implementation does no useful work.

The first optimization seeks to hide some of the latency of performing these cache accesses by doing other processing before spinning for the result. We can update the directory, advance the transfer state, and prepare the headers of the message, all under the assumption that the cache later responds to indicate success. This work is *speculative*, however, because in the unlikely event of

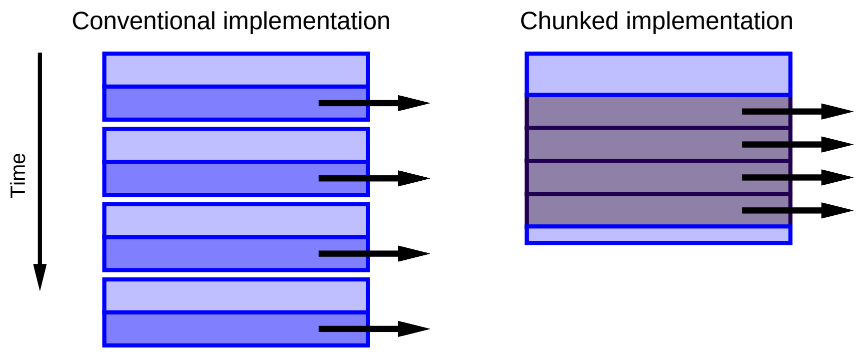


Figure 5.6: Potential benefit from chunking. The benefit derives in part from sharing start-up overheads between several lines. Light shaded blocks represent handler start-up overhead, dark shaded blocks represent time taken processing actual transfer lines, arrows indicate messages leaving the node for the destination.

a simultaneous writeback (described earlier), the cache access fails. In that case, the handler must “roll back” its state modifications to undo the work done under this incorrect assumption.

Amortizing Transfer Overhead

Earlier we mentioned a technique called *chunking* which allows the transfer handler to send multiple lines in a single invocation. Chunking allows us to amortize the software overheads incurred each time the handler starts up. This optimization is not applicable to the destination side because there the transfer is still observed as a series of individual lines. Figure 5.6 illustrates how amortization may improve performance. Even though the overhead of the chunked implementation may be higher, as shown, sharing the overhead is ultimately beneficial.

To implement chunking, we use software pipelining to send multiple lines within a loop. Pseudo code for the chunked handler appears in Figure 5.7. Software pipelining provides several important benefits in this handler. First, we build on the cache access optimization described above: chunking allows the PP to perform speculative processing on the current line as well as the initial processing for the *next* line while waiting for the cache response. An additional benefit is that some key transfer state can be kept in registers throughout the multiple lines. We set the maximum number of lines to send in one burst, called the *chunk size*, though the handler may need to send fewer lines if outgoing network queues fill.

Our implementation of the chunked transfer handler uses a carefully designed variable called the *chunking word*. This word contains a packed bit field array that holds key information for each of the lines in the chunk: its MAGIC buffer number and an indication of whether the line was requested from the cache. This allows us to advance the state for the software pipeline merely by shifting the chunking word. Note that the bottom of the main loop checks the result for a cache access in the previous iteration; this check can be done merely by consulting different bit field offsets in the

```
void MemCpyChunkedFullAlignedTransfer()
{
   Calculate maximum allowed chunk size;
   Preallocate buffers up to chunk size;
   Determine actual chunk size based on buffer availability;
   Read transfer state and address pointers;

   Initialize chunking word;

   while (Lines remain in chunk) {
      if (Line busy) {
         Set retry bit;
      } else if (Line Dirty) {
         Form request for cache intervention;
         Request line from cache;
         Mark line in chunking word as requested;
         Speculatively update directory;
      } else { /* Clean */
         Read line from memory;
         Form header for destination;
         Send message;
      }

      if (Previous iteration marked requested in chunking word) {
         Wait for cache controller reply;
         if (Request Succeeded) {
            Write line to memory;
            Send line to destination;
         } else {
            Roll back directory state;
            Mark retry bit;
         }
      }

      Shift chunking word;
      Advance transfer state;
      Check for destination page crossing;
      Decrement chunk lines remaining;
   }

   Handle final dirty check (identical to above)
   Write back transfer state

   Update remaining count;
   if (Transfer done) {
      if (Lines experienced problems) {
         Switch to cleanup mode;
         Reschedule;
      } else {
         Unschedule;
      }
   } else {
      Reschedule;
   }
}
```

Figure 5.7: Pseudo code for the optimized implementation of the locally coherent transfer handler

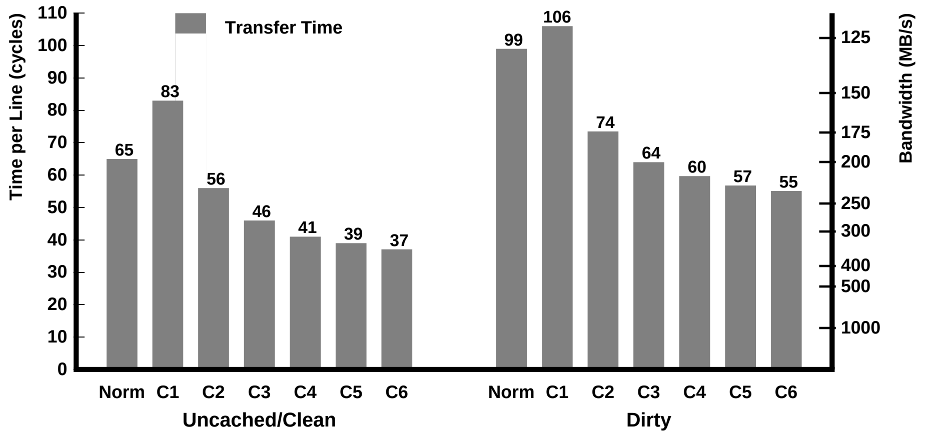


Figure 5.8: Transfer handler performance for different chunking and caching parameters. The x-axis labels indicate the maximum chunk size in cache lines and source buffer caching state (Uncached/Clean or Dirty). The y-axis and bar labels show the duration in MAGIC cycles. "Norm" shows the performance of a non-chunked version.

chunking word. By preallocating the data buffers for the handler, the software pipeline never runs out of buffers mid-handler, but instead knows *a priori* how many buffers are available and can plan the pipelining accordingly.

Unlike the handler which sends only one line, the chunked handler does *not* abort its work when it encounters a busy line or a failed cache access. Since the other lines are independent they may still succeed, so the handler instead forges onward to process the other lines, marking the failed lines for retry by the cleanup handler. An advantage of this approach is that it allows more time before the line is retried, which gives the transient case more time to resolve itself. If we retried right away, the line might fail again for the same reason. The downside to this greedy approach is that processing skipped lines later is slightly slower and cannot benefit from chunking. For example, using chunking (at a chunk size of 4 lines), it takes 43 cycles for each uncached line and 62 cycles for each line that is dirty in the processor cache. The retry handler, which processes lines individually takes 64 and 96 cycles to handle these same cases.

Figure 5.8 shows the performance of the transfer handler as a function of the chunk size. We show two different source buffer caching states: uncached or dirty local. This plot also shows the performance of the unchunked handler ("Norm"). As we might expect, chunking adds overhead to the handler, so the unchunked version is faster than the 1-line chunked version. As the chunk size increases, however, we see clear benefits from amortizing the overhead. These benefits show diminishing returns around 4–5 lines. Amortizing overhead does have its cost: cache misses during the transfer experience longer latency because the chunked transfer handler runs longer without interruption than does the unchunked version. Figure 5.9 shows the total duration of the handler,

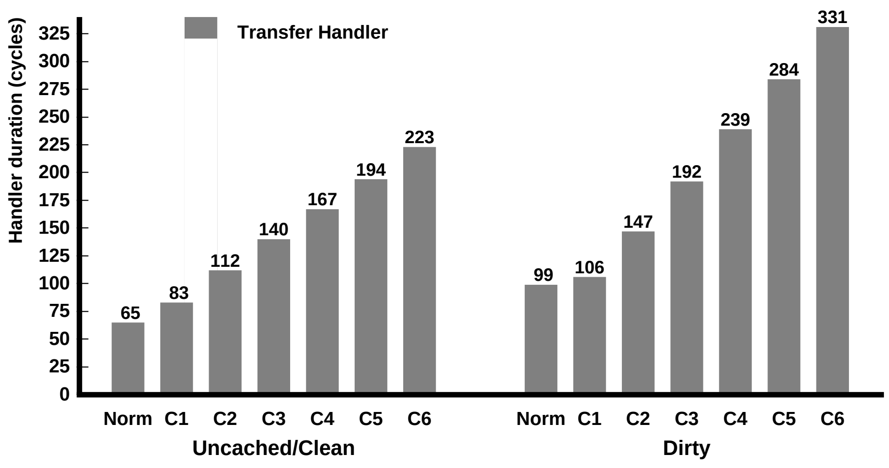


Figure 5.9: Total transfer handler duration for different chunking and caching parameters. The bars are labeled as in Figure 5.8.

which quantifies the potential for increase in cache miss latency. In practice, the best approach is to balance these two effects. We use a chunk size of four in our experiments since this provides gains from amortization while maintaining some degree of fair access to the PP.

5.2.4 Unaligned Transfers

In the simplest case, the data involved in a memory copy has the same offset within cache lines at its source and destination. Such a transfer can be carried out by moving entire aligned cache lines of data.[4] However, if the destination address has a different cache line offset than the source, each data word must change its position *within the cache line* (by a distance related to the difference in the offsets). One solution is to perform this function in PP software by loading a cache line then realigning its data words individually. In the PP, this realignment is prohibitively expensive, adding a minimum of 3 cycles per doubleword (48 cycles/line) to the transfer time.

To improve performance in this case, we introduced a hardware mechanism in FLASH (described in Section 2.3.4) that allows us to adjust alignment more efficiently. Figure 5.10 shows an example block transfer illustrating how the mechanism we provide can adjust the alignment of words within a line. In the example, the transfer begins and ends in the middle of cache line boundaries (panels 1 and 2). The same figure shows (panels 3 and 4) how the alignment would be changed during the transfer for a cache line aligned destination buffer.

One of the insights we provide by this research is showing that block transfer support can be efficient in FLASH using line-at-a-time transfers, and that we can effectively leverage the existing

[4]The only exceptions are the beginning and end of the block, which may be partial lines. We handle these boundary conditions in the fbcopy library in the processor, since they are not performance-critical.

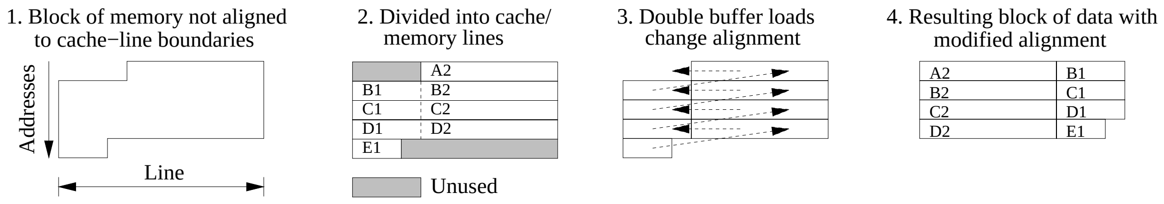

1. Block of memory not aligned to cache−line boundaries
Addresses
Line
2. Divided into cache/ memory lines
A2
B1
B2
C1
C2
D1
D2
E1
Unused
3. Double buffer loads change alignment
4. Resulting block of data with modified alignment
A2
B1
B2
C1
C2
D1
D2
E1

Figure 5.10: Support for arbitrary block transfer alignment.

support for data movement. The ability to change alignment of data, which extends high performance to nearly all cases, is the only hardware feature we added to MAGIC for memory copy.

For full generality, each interface would have to support double buffer operations to the data buffers, and would allow any alignment down to byte granularity. In practice, we expect the vast majority of block transfers to require realignment only down to the 32-bit word size, since this corresponds to the smallest integer data type used in most applications and the smallest floating point data type available on the R10000. To reduce the hardware impact in MAGIC of providing this feature, we provide double buffer support only from the memory interface.

Given this latter restriction supporting coherence for unaligned transfers requires that we first collect any lines of the transfer data which are dirty and write them back to memory. We perform this collect phase using a bit mask nearly identical in function to the one used for retry. In fact, we use the same bit mask, but interpret it in a slightly different way. We initially mark *all* the lines of the message as needing collection, then we iterate once over the body to gather lines which are dirty in the cache, clearing the bit for each line that succeeds.

Once the entire block is valid in memory, we switch to a special version of the transfer handler designed especially for unaligned transfers. Since we have already collected the lines, this handler is guaranteed that the copy in memory is up-to-date and can *ignore the directory state altogether*, focusing instead on moving data efficiently. As a result, the transfer phase is actually faster than in the normal case. As described in Section 2.3.4, the alignment hardware requires two memory loads to fill a data buffer. If we generate only one realigned line, part of each load is wasted (i.e., 2 memory loads are needed). However, if we generate multiple unaligned lines back to back, the overflow from the second load forms the initial load for the third (and so on). Thus we only incur one wasted load for the block (i.e., $n + 1$ memory loads are needed for n generated lines). For this reason, unaligned loads can show even more gains from the chunking optimization described earlier. Figure 5.11 shows the performance of unaligned transfers as a function of chunk size. Note that the performance of the transfer phase is independent of caching state because the lines are always read from memory, while the collect phase increases in cost when lines are dirty. To keep code size down the collect phase does not take advantage of the ability to software pipeline—its overhead could be reduced somewhat by expanding the code.

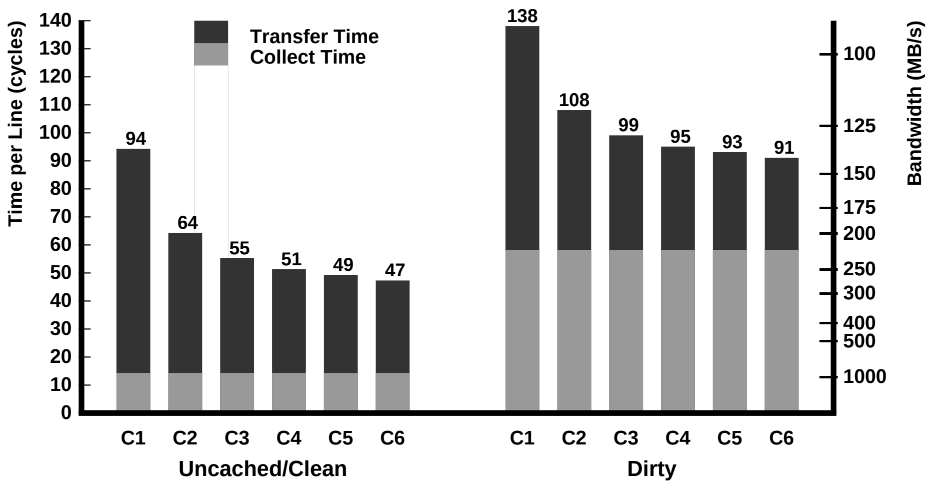


Figure 5.11: Unaligned transfer performance as a function of chunk size and caching state.

5.3 Fully Coherent Transfer Model

While a number of previous systems have provided support for block transfer protocols [KA93, BLA+94, RLW94, ACD+95, FAB+96], none addressed full integration with cache coherence. These systems advocate the local coherence model described in the previous section which constrains the caching of source and destination buffers to their respective local nodes. However, this model breaks down in a shared memory system running a modern operating system such as IRIX in which processes may migrate to improve load balancing. In such an environment, assumptions about the local coherence and local allocation of pages no longer hold; only a block transfer protocol without such restrictions is widely usable. More recently, several systems have begun to provide coherent block transfer, especially for use by the operating system [NAB+95, LL97, WGH+97].

In this section, we describe the extensions to the protocol to provide full integration with cache coherence. This entails both of the issues described in Section 5.1.1: supporting coherence of message data despite remote caching, and supporting pages with remote homes.

5.3.1 Cache Coherence for Block Transfer Data

Providing efficient and transparent cache coherence is a key aspect of making primitives such as block transfer easy to use in a shared memory environment. Since the source and destination data may be cached anywhere in the system, the block transfer protocol must incorporate a subset of the cache coherence protocol to efficiently obtain the latest data and maintain coherence of the source and destination buffers. Figure 5.12 illustrates the caching scenarios we might encounter, using a representation originally used by Lenoski [Len92]. We focus on portions of this figure to illustrate the coherence solution. The description assumes that the data transfer occurs directly from the

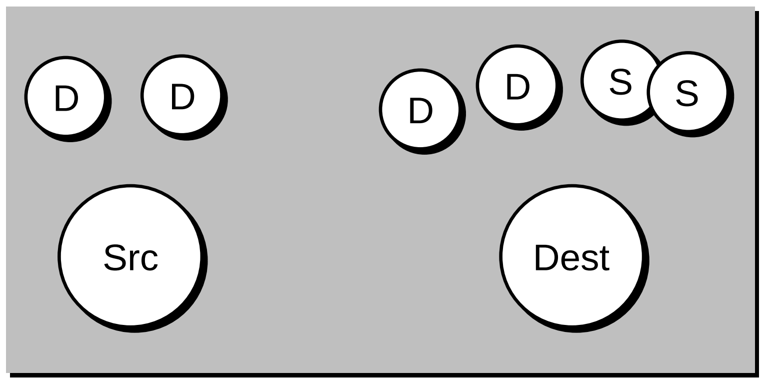


Figure 5.12: Example coherence scenario. This scenario illustrates the remote cases a fully integrated protocol must support. The two large circles represent the source and destination nodes of the transfer. Other circles represent lines of the transfer cached remotely, either in shared state (S), or dirty (D).

home node of the source buffer to the home node of the destination buffer; Section 5.3.2 describes the general case.

Coherence for Source Buffer Data

There are three cases to consider for coherence of the source buffer: the data is clean, the data is dirty in the local processor's cache, or the data is dirty at a remote node. If the source node has a valid copy of the data, it can be retrieved from the cache or memory as described previously. However, when the data is dirty at a remote node, the task of retrieving the valid data requires remote communication. The simplest solution is to pause the transfer, retrieve the line from the remote node, send it to the destination, and then proceed with the transfer. Though straightforward, this approach fails to exploit any parallelism in retrieving the data for multiple lines.

A more efficient solution is to accomplish the transfer in two distinct phases, one that collects the dirty data of the source buffer from remote nodes and one that sends the data to the destination [KA93]. This technique, called *collect and send*, allows the collect phase to retrieve multiple lines simultaneously (similar to the way we implemented unaligned transfers in the previous section). For a machine such as FLASH that performs transfers a cache line at a time, it is also possible to pipeline the collect and send phases instead of keeping the phases distinct. In this approach, when the handler detects that a line is dirty on a remote node it requests the line and proceeds with the transfer. The response eventually arrives back at the source node and triggers a handler that sends the line to the destination and writes the data back to memory. We refer to this as *pipelined collect and send*.

Still another alternative is to send the data directly from the dirty remote node to the destination node (similar in philosophy to the optimization in DASH for the three-hop dirty remote

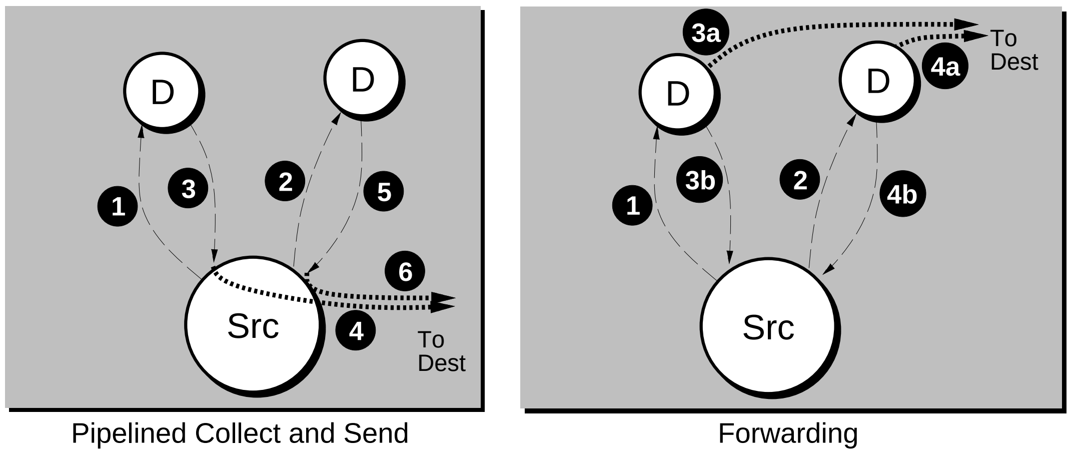


Figure 5.13: Source side coherence alternatives. In both cases the source node sends fbcopy messages to fetch dirty cache lines from the remote holders

case [LLG+90]). We refer to this as the *forwarding* approach. The request from the source node to the dirty remote node carries a single word of data that specifies the destination address. The dirty node forwards the data to the destination node and sends a writeback to the source node. Figure 5.13 illustrates these two alternatives.

Compared to pipelined collect and send, forwarding reduces protocol overhead since the source node can perform an ordinary writeback without remapping the message for sending to the destination node. On the other hand, there may be higher occupancy at the network interface of the dirty node since forwarding leads to two outgoing messages (the writeback to the source node plus the forwarded message to the destination node), compared to just one message for the collect and send case. The total traffic is the same, but since forwarding sends both messages from the same network port the messages are serialized. Thus the choice between the two approaches depends on the relative speeds of the network and the protocol processor. As we show in Section 5.4.3, the forwarding approach is superior for network bandwidths of 400 MB/s per link or greater, assuming a 100 MHz PP.

Coherence for Destination Buffer Data

The destination node must in turn coherently modify the data in the destination buffer, which requires invalidating all stale copies of the data. The handler that receives the line writes it into the destination buffer in memory and then eliminates any copies in processor caches. Figure 5.14 shows the actions of the protocol. If there are any clean copies of the line, the handler sends an invalidation request to each copy.

Like the locally coherent model, we accept lines which arrive at the destination even if they are cached dirty. In the fully coherent model, this support is extended to handle lines cached *remotely*

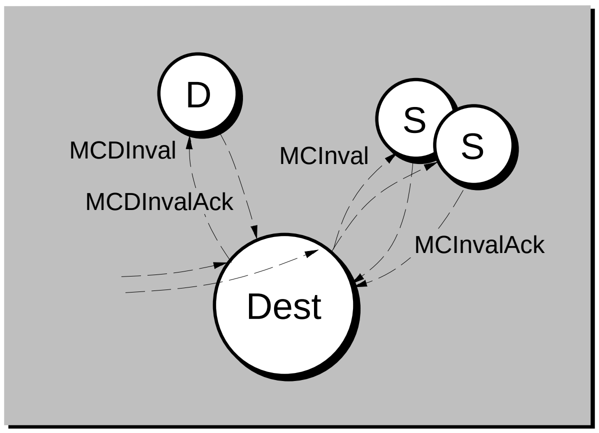


Figure 5.14: Destination side coherence. The destination sends invalidations to nodes that have a destination line cached dirty (D) or shared (S).

as well. In this case we send the dirty invalidation to the remote node, telling its cache to discard its dirty copy of the line without writing it back. Similar to the locally coherent model, there is a possible race between the arrival of the invalidation message and a writeback of the line from the remote cache. If a writeback occurs before the invalidation arrives, the writeback overwrites the block transfer data that has been written to memory. To detect this race, the invalidation message checks the response from the remote cache. If the line has already been written back, a negative acknowledgment message is sent back to the destination node, triggering a retry of the line from the source.

5.3.2 Support for Remote Homes

So far we have assumed that the transfer is carried out by the initiator node. However, as described in Section 5.1.1, the initiator node may be distinct from the home nodes of the source buffer pages. Since the cache coherence protocol must consult the directory state to determine the appropriate coherence action, the initiator must communicate with the home for each line of the transfer. The naïve approach of requesting each line individually leads to increased traffic in the network and prevents the home node from leveraging the chunking optimization to increase performance. In addition, this implementation is substantially more complicated, since it introduces many more special cases and race conditions.

A better approach is to delegate the transfer to the home node and have it carry out the transfer on the initiator's behalf. After delegating the transfer to the home node, the protocol behaves exactly as if the block transfer were initiated there. The only difference is that a completion signal is sent from the home node back to the initiator node when the transfer is complete. Despite an overhead of approximately 2.5 μs to delegate the transfer, this approach leads to better overall performance

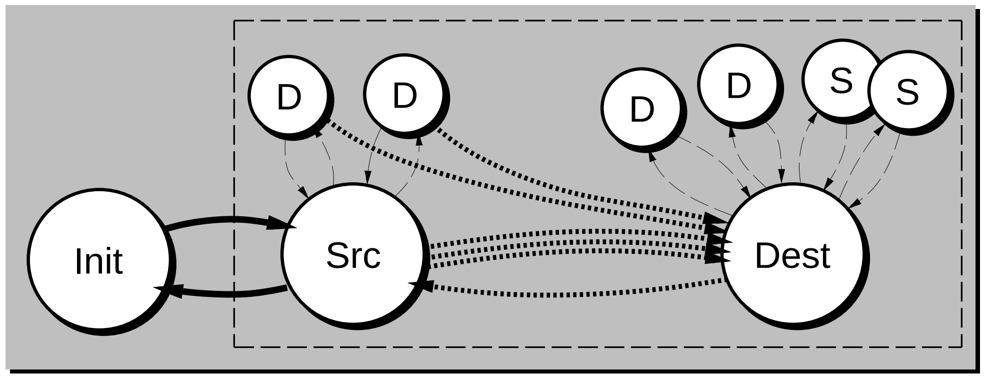


Figure 5.15: Complete fbcopy protocol including remote home support. Transfers initiated at a node other than the home of the source buffer are delegated to that node, as shown. The actions inside the dashed box are present even when remote initiation is not required.

for transfers longer than a few lines. Figure 5.15 illustrates a remote home scenario, showing how it can build on the existing implementation.

5.3.3 Implementation Issues

During the implementation of the protocol, we made some design decisions and observations that are important to mention. We briefly describe these issues below:

Copies within a node. Our goal in providing block transfer is to accelerate copies between two nodes, so we do not provide support for copies in which both buffers reside on the same node. Since local memory access is highly optimized, our experience suggests that the processor is more appropriate for these *intranode* copies. During transfer initiation, the protocol processor checks the pages involved in the transfer to see if they are located on the same node. If so, initiation returns a result code indicating that the processor should perform the copy; the library then calls the normal bcopy routine. By encapsulating this functionality in the fbcopy library call, the user can simply invoke fbcopy without regard for page placement.

Remote home parallelism. Though relatively unlikely, it is possible that the source pages of a transfer are not only remote, but scattered among several nodes. Given the nature of remote home transfers, there is nothing to prevent us from initiating multiple remote home transfers *simultaneously*, to increase the overall parallelism of the transfer. We have chosen not to implement this feature for two reasons. First, this would significantly increase the complexity of managing remote home transfers. Temporary resource shortages can cause any of the remote home initiations to fail; by having only one remote initiation at a time, we reduce the number of failure scenarios. Second, since it is fairly likely that the destination pages are collocated, having multiple simultaneous remote home transfers may cause an undesirable hot spot at the destination.

Protocol code size. The total amount of handler code added for the fbcopy protocol is about 19 KB. In contrast, the base cache coherence protocol in FLASH is about 44 KB. Although the block transfer protocol code is only about 40% of the cache coherence protocol in size, it is much larger than we had initially expected. As described in Section 4.4, cache coherence and block transfer have similar, but slightly different needs. We could have added special cases in the coherence protocol to promote code sharing, but since this would slow down cache coherence it was deemed unacceptable.

5.4 Low-level Performance Analysis

In this section we consider the performance of the fbcopy protocol at a low level. We begin by describing the simulation environment used in our experiments. We then provide a detailed analysis of the block transfer protocol to illustrate the various factors that determine the overall performance of this protocol on FLASH.

5.4.1 Simulation Environment

The simulation environment used in this study provides a complete model of the FLASH system, allowing us to accurately study its performance running a variety of workloads. We use the FlashLite memory system simulator, described in Section 2.4.2, which models the performance of MAGIC in detail. Simulation of the main processors and I/O system is provided by the SimOS environment [RHWG95].

The actual R10000 processor runs at 200 MHz, issuing up to four instructions per cycle. Unfortunately, we do not have a simulation model of the R10000 processor. Instead, we estimate its performance by using a single-cycle processor model running at 400 MHz. Assuming the R10000 is able to sustain only a fraction of its peak issue rate, this should at least approximate its performance. We assume a two-level processor cache hierarchy. The first level consists of split 32 KB instruction and data caches, both 2-way set associative with 64 byte lines. The second level is a unified 1 MB cache, 2-way set associative with 128 byte lines. The time to satisfy a first level cache miss (second level cache hit) is 50 processor cycles.

We simulate the MAGIC chip running at its target frequency of 100 MHz (for consistency in our discussion, "cycles" *always* refers to 10 ns MAGIC cycles). The memory system provides MAGIC with the first 64-bit word of a cache line in 14 cycles, followed by an additional word on each successive cycle (i.e., total of 15 extra cycles). Memory can be accessed with a new address during the transfer stage. Accessing data in the processor's cache from MAGIC takes longer than accessing main memory because of the required processor intervention. In this case, the cache responds with the state of the line after 15 cycles and with the first data word 5 cycles later, followed by 15 cycles of transfer time for the remaining data words. Unlike memory, accessing the next line of data from the processor cache is delayed until the previous transfer phase completes. As a comparison point versus cache coherence, the latency for cache-coherent reads in FLASH is approximately 27 cycles for local misses and 111–191 cycles for remote misses (larger latency corresponds to the data being dirty at a third node).

For this chapter, we model a network with a bandwidth of 400 MB/s, of which 320 MB/s is usable (the rest is consumed by header overhead).[5] Here we also assume an average network latency of 22 cycles, which is appropriate for a small FLASH system requiring an average of three hops.

In the actual FLASH system, the PP instruction cache is only 16 KB. As noted in Section 5.3.3, the complete block transfer protocol is approximately 19 KB, and cache coherence adds another 44 KB. If we simulated with the actual instruction cache size, the results would be dominated by cache effects. Instead, we perform this and our other studies with a very large instruction cache to allow us to isolate the fundamental issues in the protocol, and not hide them behind an implementation constraint of the initial FLASH system.

The fbcopy protocol and the base cache coherence protocol are specified in C and compiled to PP instructions as described in Section 2.4. The results presented in this dissertation are based on a cycle-accurate emulation of the actual protocol code. Even though the automatically generated code is fairly efficient, higher performance can be achieved by hand tuning the compiled code—we address this issue in Section 5.5.1.

5.4.2 Transfer Overhead

We divide the time to perform a data transfer into two parts: *(i)* a fixed overhead portion, and *(ii)* a variable transfer time proportional to the size of the message. In this section we focus on the overhead portion; the performance of the transfer portion is considered in detail in the next section. Recall that Figure 5.2 provides a high-level illustration of the transfer. Figures 5.16 and 5.17 (explained in the next section) illustrate the transfer overhead more concretely including the data presented below.

The total protocol overhead in fbcopy is about 677 MAGIC cycles or 6.8 μs. This represents all the fixed costs occurring during the transfer that cannot be attributed the transfer part itself. Table 5.1 explains the breakdown of this overhead by task. Note that the main processor is not required to wait during the entire verification and startup phases listed in Table 5.1. After the PPC

[5] The final version of the FLASH system has switched to use the CrayLink interconnect, so the actual network bandwidth in FLASH will be higher.

Table 5.1: Breakdown of fbcopy overhead components, expressed as 10 ns MAGIC cycles.

Phase	Overhead component	Cycles
Initiation/	Initiation command processing	131
Verification	Transfer set-up and command verification	233
Startup	Envelope exchange with destination	102
	Initial network latency	18
Completion	Receiver cleanup at completion	16
	Final network latency	18
	Cleanup and completion flag update	159
Total overhead		677

has been verified, (approximately the first three microseconds), the processor can begin working in parallel.

The high overhead in this protocol arises for several reasons. First, the fbcopy protocol accepts several parameters from the application and MAGIC must first verify them to protect the system. Second, the set-up phase both initializes the transfer state and performs some calculations once so that later the transfer handler can execute more quickly. Finally, updating the completion flag coherently with PP ownership accounts for approximately 1 μs of the final handler, as described in Section 4.4.

This amount of protocol overhead is not fundamental to MAGIC itself, but rather it depends primarily on the protocol characteristics. This implementation especially targets large message sizes, for smaller messages a more lightweight approach would be used. We show in Chapter 6 that MAGIC is able to support synchronization protocols at very low overhead through both careful design and by providing a more lightweight interface than is possible in memory copy.

Furthermore, the complete overheads in traditional message passing machines may be higher due to the cost of initiating a message reliably and with protection. One lightweight protocol implemented on top of the Intel Paragon incurs over 18 μs latency for a short message when all necessary protection checks are used [BSSD96]. The Cray T3D requires approximately 6 μs of processor involvement merely to set up a transfer, despite having a dedicated block transfer engine. Our protocol mitigates some of these set-up costs through the processor-MAGIC interface techniques described in Chapter 4.

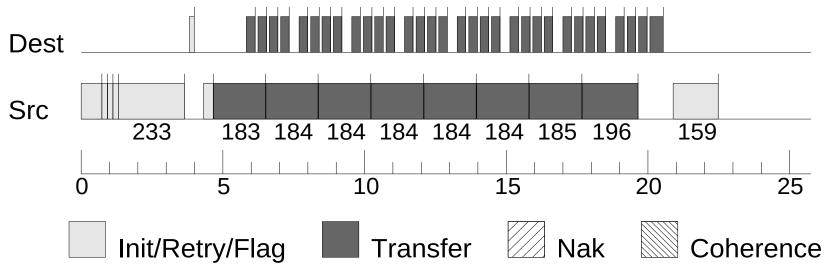


Figure 5.16: Time line of a two-node block transfer. Source and destination buffers are uncached. The axis shows time in microseconds. Each block is a single handler running on MAGIC (long-running handlers also show their execution time in 10 ns MAGIC cycles.)

5.4.3 Transfer Bottlenecks

The performance of the block transfer protocol is determined by one of three possible bottlenecks: source side processing, destination side processing, or network bandwidth. To illustrate these factors, Figures 5.16 and 5.17 show timelines of the handler occupancies for block transfers in two different caching scenarios. The results are from the simulation of a 4 KB transfer initiated at the source node, using the forwarding protocol and a maximum chunk size of 4 lines. We have divided the handlers into four categories: *(i)* those that handle initiation, retry, and completion flag update (labeled Init/Retry/Flag); *(ii)* those that send and receive the data messages (Transfer); *(iii)* those that handle coherence for remotely cached copies (Coherence); and *(iv)* those that process corner case negative acknowledgments (Nak). We use thin vertical lines to indicate the end of each handler and show the duration in cycles for handlers that last for more than 100 cycles.

Figure 5.16 shows a transfer in which both source and destination buffers are uncached. The initiation phase consists of four uncached writes followed by an uncached read. The time line shows the initiation phase on the source node followed by the communication of the "envelope" to the destination, lasting about 3.5 μs. Once the destination node replies, the source node repeatedly invokes the chunked transfer handler, each time sending 4 lines (thus 8 invocations are needed to send the entire 4 KB). The destination receives each line individually, acknowledging the source when the transfer is complete. The total time for the transaction is about 22.5 μs, with the actual transfer occupying about 16 μs. This corresponds to peak transfer bandwidth of about 276 MB/s and an average bandwidth (including initiation and completion overheads) of 182 MB/s. Note the gaps between invocations of the destination handler due to the faster processing of lines at the destination.

The time line in Figure 5.17 shows a transfer using the forwarding protocol described in Section 5.3.1 in which the source buffer is cached dirty at a remote node and the destination buffer is uncached. For each line, the transfer handler sends a request to the dirty node to forward the line directly to the destination. The dirty node also sends a writeback to the source node (visible as the

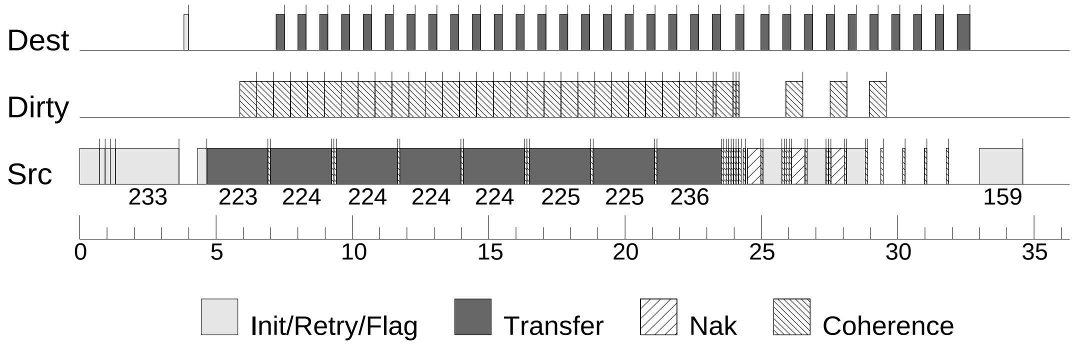


Figure 5.17: Time line of a block transfer with the source buffer dirty on a remote node. In this case, queue limitations cause negative acknowledgements, forcing three lines to be retried.

short handlers on the source). The network is the bottleneck in this case: the dirty node's protocol processor sends messages faster (two messages every 62 cycles) than the network can absorb them (one message every 40 cycles). This causes the outgoing network queue to fill three times during the transfer. Each time, the handler detects the full outgoing queue and sends a NAK to the source (visible on the dirty node in Figure 5.17 as the three short handlers between 23–25 μs). MAGIC retries these requests successfully (between 26–30 μs on the dirty node). The increased coherence handling in this transfer reduces the peak transfer bandwidth to 146 MB/s (or 118 MB/s including the overhead). We previously described our approach to provide flow control for the receiver. This example indicates that future research may wish to consider some degree of sender flow control as well.

As discussed in Section 5.3.1, the decision to use forwarding instead of pipelined collect-and-send (PC&S) depends on the relative cost of protocol processing and network bandwidth. Figure 5.18 shows the simulation results for these two alternatives in FLASH, varying only the network bandwidth and source caching state. For a slow network, the network overhead of the extra messages in the forwarding approach is greater than the protocol processing overhead incurred by PC&S. At higher bandwidth, these extra messages become less expensive and forwarding becomes more efficient. In this simulation, the crossover point is between 200 and 400 MB/s. Since we model a 400 MB/s network, we use the forwarding approach. Of course, in cases where severe contention decreases effective network bandwidth, the PC&S approach may be more appropriate.

So far we have discussed two specific caching scenarios for source and destination side buffers. Figure 5.19 shows the performance for other possible situations. This figure indicates whether the performance limit in each case is network throughput, source side handlers, or destination side handlers. As an example, 46 cycles for the uncached/clean case on the source side corresponds to one fourth of the chunked handler occupancy of 184 cycles (shown in Figure 5.16). For a given caching state, the peak transfer rate is limited by the slowest of these three components. For the destination

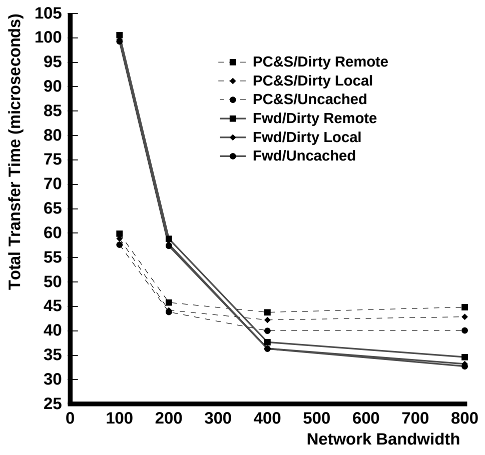


Figure 5.18: Comparison of forwarding (Fwd) vs. pipelined collect-and-send (PC&S) for different source buffer caching states.

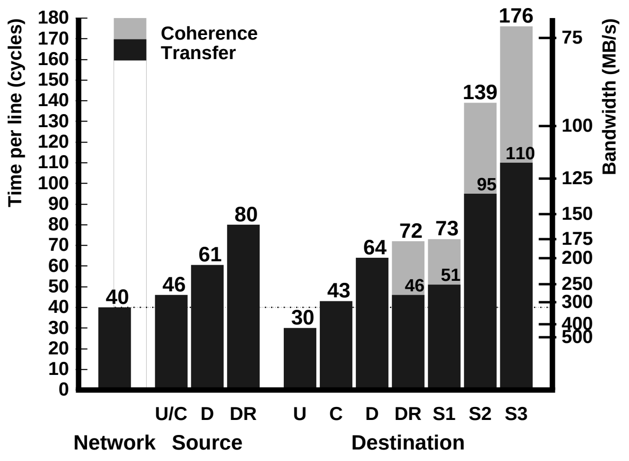


Figure 5.19: Bottlenecks in fbcopy from the network, source protocol processing, and destination protocol processing. The bottleneck incurred is given by the slowest of the three components, as appropriate for the caching state of the buffer: uncached (U), cached (C), dirty (D), dirty remote (DR), clean with 1–3 sharers (S1–S3).

side, the coherence portion of the bar shows the time to handle invalidation acknowledgments from remote copies. One interesting effect is the sudden jump in destination node transfer handler cost for two or more remote sharers (between the S1 and S2 bars in Figure 5.19). This is an artifact of our cache coherence directory format, which stores the first sharer in the main entry but places subsequent sharers on a linked list. The remote sharer cases are also impacted by the increasing coherence overhead of 22 cycles per invalidation acknowledgment.

5.5 Benchmarks and Performance Comparison

In this section we extend the results in the previous section by studying the fbcopy protocol in the context of microbenchmarks and applications. In these simulations we compare fbcopy with processor copy—we consider both regular bcopy and bcopy augmented with prefetch instructions [MG91]. Prefetched bcopy can be very effective in FLASH due to its low latency network and highly optimized memory system, including a processor with an aggressive prefetching implementation. FLASH also contains long (128 byte) cache lines, which can achieve many of the benefits of block transfer [WSH94].

In the first section, we study the block transfer implementations without operating system effects. In subsequent sections we show experiments using the IRIX Version 5.3 operating system from Silicon Graphics, Inc., which we modified to use fbcopy.

Table 5.2: Transfer latencies to send 4 KB using bcopy, prefetched bcopy, and fbcopy, expressed in microseconds. The network traffic for the different cases is also listed. Buffer caching states are: uncached (U), shared local/remote (L/R), dirty local/remote (DL/DR).

	Source Caching State	Destination Caching State: Normal PP U/L	Normal PP R	Faster PP U/L	Faster PP R	Network Traffic
Bcopy	U	65.8	67.6	62.2	62.5	8 KB
	DL	47.7	52.0	45.0	45.4	8 KB
	DR	105.0	106.8	98.1	98.3	12 KB
Pref Bcopy	U	28.8	30.0	26.5	27.3	8 KB
	DL	20.5	24.0	19.5	21.0	8 KB
	DR	37.3	38.6	35.2	36.1	12 KB
Fbcopy	U	23.3	33.4	19.0	21.1	4 KB
	DL	28.7	33.7	19.2	21.3	4 KB
	DR	36.7	37.7	33.7	34.6	8 KB

5.5.1 Microbenchmark Analysis

We first compare the performance of fbcopy against the other transfer primitives by simulating the transfer of a 4 KB page between two nodes. As described in the previous section (and as shown in Figure 5.19) the performance of coherent block transfer depends on the caching state of the involved lines; we focus on several cases which tend to be more common in applications using block transfer. For simplicity, we assume all the lines of a particular buffer are in the same caching state. The transfer is executed at the home of the source buffer.

There is an additional caching effect which impacts processor copy since it must bring the buffers into the cache. If this causes spills—which is likely—writebacks or replacement hints are generated, which may increase the latency of the block transfer operation. To model this effect, we first warm the caches with an equal mix of clean and dirty lines then cache the transfer buffers as appropriate for the experiment. Since fbcopy does not bring the buffers into the processor cache, it is immune to this effect.

Normal Protocol Processing

Table 5.2 shows the time required by the source node to perform the transfer in the scenarios described above. Fbcopy significantly outperforms standard bcopy, executing the transfers more than twice as fast in most cases. Bcopy incurs expensive cache misses to copy the data, which poorly utilize the processor during the transfer. Prefetched bcopy largely avoids the miss penalties of bcopy by acquiring the lines in advance of their use and by fetching multiple lines in parallel. This technique is quite effective in improving transfer performance, so much so that in half the cases prefetched bcopy finishes the transfer faster than fbcopy. Notably, prefetched bcopy is faster when the source buffer is dirty in the cache, since fbcopy incurs higher overhead to request the data from the cache.

However, processor copy techniques (including prefetched bcopy) tie up the processor for the entire duration of the transfer, rendering it unable to perform other work. By delegating the block transfer to MAGIC, fbcopy frees the processor to continue working while the transfer proceeds. The effectiveness of the processor working in parallel will vary based on its use of already-cached data. Miss-limited applications may or may not show gains as a result of increased miss latencies due to the PP being occupied by the transfer.

Fbcopy also uses the network more efficiently. Processor copy fetches the entire destination buffer even though it is immediately overwritten. In contrast, since fbcopy does not acquire ownership of the destination buffer at the source, these lines never cross the network. Instead, fbcopy only incurs network traffic for the data actually being transferred. The rightmost column of Table 5.2 shows the total traffic caused by the different implementations for a 4 KB transfer.

Consumption of Transfer Data

Though transfer time is important, the performance of block transfer is not based solely on that factor. It may also include the time needed to “consume” the transfer data at the destination node. To quantify this effect, we simulated reading the values in the buffer at the destination node after the transfer.

Since the source node performs the transfer, in processor copy the destination buffer ends up in the source node’s cache. When the buffer is consumed, the destination node experiences dirty remote cache misses. Reading the buffer in this case takes 70 μs. Because fbcopy deposits the data in the memory of the destination node, only local misses are taken. As a result, the consumption phase after fbcopy lasts only 20 μs. Furthermore, in processor copy the data involved in the transfer may replace unrelated lines still in use by the source node processor. This effect is particularly significant in the primary cache, which can be completely filled by a mild-sized block transfer. The impact of this cache pollution is application dependent, based on the amount of subsequent reuse of the replaced lines. As an indication of the potential cost, it takes approximately 32 μs to refill the 8 KB of data displaced during a 4 KB processor copy (longer if the memory is remote).

Faster Protocol Processing

We also considered the effect of faster protocol processing to determine if the performance of fbcopy is fundamental to the architecture. We altered our simulator to execute protocol code twice as fast, allowing us to quantify the potential for increased performance due to protocol code optimizations or a more aggressive protocol processor implementation.

The results are shown in Table 5.2. They show mild gains in most cases for processor copy, but more significant gains for fbcopy. This can be explained by looking at the performance limits imposed by the different resources in the system. With faster protocol processing, cache misses are serviced more rapidly, which improves the performance of processor copy. However, in the base coherence protocol in FLASH, the PP is only a few cycles slower than the memory system in the common miss handlers so these gains are small [HGDG94, KOH$^+$94]. Fbcopy, in contrast, generally performs more protocol processing per line. As a result, there is more potential for gains from faster processing. While processor copy sees only 6–13% benefit from faster protocol processing, fbcopy gains between 8–36% in these benchmarks.

5.5.2 Message Passing Interface (MPI)

To study the performance of block transfer operations in a more realistic context, we examine message passing primitives from the Message Passing Interface (MPI) standard [Mes93]. We use version 1.0.11 of MPICH from Argonne National Laboratory [GLS]. MPICH supports several different underlying physical transport mechanisms, including one designed for shared memory multiprocessors such as the SGI Challenge. This code runs without modifications in the SimOS simulation environment.

In the shared memory implementation of MPICH, bcopy is used to perform the data transfer of large messages (2 KB or greater). MPICH uses IRIX's support for allocating a shared memory region between processes, while the rest of the application address spaces are private, consistent with the distributed memory model of MPI. To send a message, the sender first copies the message data from its private memory into the shared buffer. The receiver then copies the data from the buffer into its private memory. We augmented the base MPI implementation to perform this operation with fbcopy or prefetched bcopy instead of standard bcopy.

We implemented a basic two-node application using standard MPI send and receive primitives in which data is communicated in a producer-consumer relationship. Figure 5.20 shows that for both normal and faster protocol processing, fbcopy outperforms both normal and prefetched processor copy over a range of message sizes. The gains from fbcopy increase with message size, due largely to the amortization of fbcopy start-up overhead.

The flexibility of the protocol processor in MAGIC enables an additional option not possible in processor copy. Since the MPI send and receive routines execute at user level, maintaining private address spaces requires two copies using the shared buffer as described above. We could avoid this limitation by implementing buffer management functionality in MAGIC, allowing the receiver to "post" its buffer to the sender in a protected fashion. With the aid of this support, the sender could use fbcopy to transfer its message directly from the sender's memory to the receiver's memory,

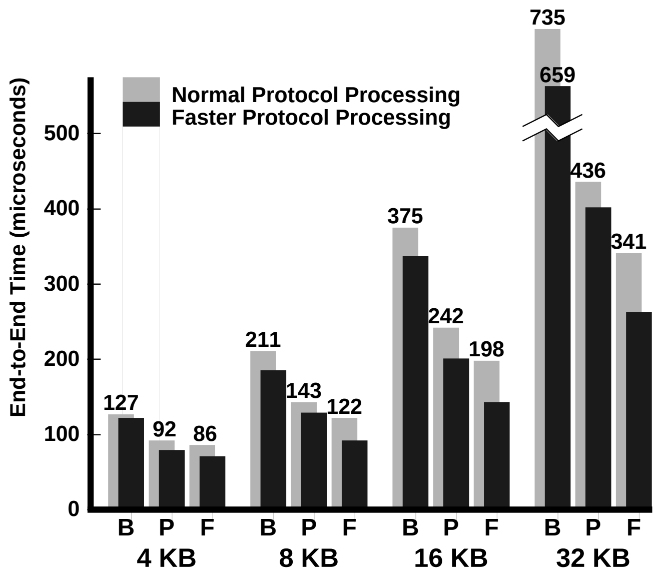


Figure 5.20: MPI transfer performance for a range of message sizes for the transfer techniques we study: Bcopy (B), Prefetched Bcopy (P), and Fbcopy (F).

avoiding the temporary buffer entirely. This technique has the potential to perform better than all two-copy implementations.

5.5.3 GNU tar Application

Since our protocol is fully integrated with cache coherence we can use it to accelerate block transfer in shared memory applications as well. In fact, one of the biggest users of block transfer may be the operating system itself. There are several kernel tasks that are primarily concerned with the movement of block data, such as uiomove which is used to move I/O data between user and kernel space. In this section, we focus on accelerating uiomove through the use of fbcopy or prefetched bcopy. Modifying the IRIX kernel was straightforward since fbcopy and prefetched bcopy are essentially drop-in replacements for bcopy. We study the impact of this acceleration on the GNU tar application performing a conventional directory copy:

```
cd $OLD; tar cBf - . | (cd $NEW; tar xBf -)
```

We simulate three nodes, two executing the tar commands and the third modeling the buffer cache in memory (we assume the buffer cache contains the files involved in the transfer). The first tar process reads data from the buffer cache through a read system call, which uses uiomove to copy the buffer cache data into the tar process's user address space. The tar process sends the data into the pipe buffer (in 10 KB chunks) using a write system call. The kernel again uses uiomove, this time to bring the data from user space into kernel space. Similar actions occur on behalf of the second tar process.

Table 5.3: Simulation results for GNU tar using different memory copy implementations.

	Normal PP			Faster PP		
	B	P	F	B	P	F
Overall benchmark:						
Execution Time	215 ms	180 ms	188 ms	202 ms	167 ms	160 ms
Speedup (vs. bcopy)	1.00	1.19	1.14	1.00	1.21	1.26
Block Transfer in use:						
% of execution time	30%	22%	27%	29%	20%	21%
CPU Occupancy for BT	65 ms	39 ms	13 ms	59 ms	34 ms	12 ms

We show the results from the tar application in Table 5.3. Both prefetched bcopy and fbcopy show speedups over the standard bcopy implementation. Though fbcopy is slightly slower than prefetching with normal protocol processing, with faster protocol processing it gains over prefetching. Furthermore, fbcopy has some advantages over processor copy in overall system resource utilization. As shown in the table, the processor copy versions require more than three times as much CPU occupancy to perform the same amount of block transfer. In the fbcopy version, this time could be utilized for parallel computation, further increasing the performance of fbcopy relative to processor copy techniques.

5.6 Related Work

As described in this chapter and Chapter 4, the goal of our message passing implementation in FLASH is to provide high performance block transfer at user level. In addition, since FLASH uses a modern operating system, we require the protocol to provide protection and coexist with virtual memory and multiprogramming. Though numerous systems have been proposed for supporting message passing protocols efficiently, most of them do not address the issue of integrating message passing with shared memory and cache coherence. Still others do not provide block transfer at user level, or fail to fully support the features needed to make them usable in the context of a modern operating system. Below we provide a comparison of our design with some of these systems.

Many systems and research proposals advocate provisions for direct user-level access to message protocols. The messaging interface is typically either *memory mapped* or *register based*. The Connection Machine CM-5 provides access to the network through a memory mapped interface [Thi91]. Register based approaches provide tighter coupling by moving the network interface into the processor and providing direct access to the interface through special registers [DFK$^+$92, HJ92, NWD93]. One of the problems with the above systems is that they are typically optimized for short messages, thus limiting the achievable bandwidth for large transfers. Another drawback is that the compute processor handles the complete transfer, thus taking cycles away from the main computation.

Several systems, such as the Intel Paragon [Int91] and *T [Bec92, NPA92], have proposed delegating message protocol handling to a second processor on the node to alleviate overheads and allow for overlap of computation and communication. Paragon uses a conventional processor that is not well integrated with the network. *T provides tighter network integration, but requires the use of custom processors as compute engines. Neither design has the ability to support cache-coherent shared memory.

The approach in the Meiko CS-2 [HM93b] is more similar to FLASH since protocol processing is delegated to a programmable network controller. However, they are not capable of supporting cache coherence protocols and they provide a *separate* DMA unit. In FLASH, the controller is optimized for efficient protocol handling and provides support for data movement such as block copy. In addition, the cost and complexity of the controller is amortized by handling both cache coherence and block transfer in a single flexible unit.

The SHRIMP system [BLA+94, BDFL96, FAB+96] from Princeton advocates the use of simple network controllers for supporting message passing style communication. SHRIMP does not provide hardware support for maintaining cache coherence. The SHRIMP philosophy is to separate protection and buffer management issues from the data transfer functionality and to only support the latter in hardware. SHRIMP provides two modes of data transfer: an explicit DMA transfer, and an implicit transfer mode that gathers uncached processor writes and sends them to other nodes at a word or block granularity. The implicit mode inherently involves substantial processor and bus bandwidth overhead, but can be provided directly at user level once mappings have been established. The DMA mode is more efficient, but may require system calls to assure protection.

The Cray T3D [Cra93] supports message passing within a single address space, but without cache coherence. The T3D supports two modes of transfer: small messages (32 bytes) that interrupt the destination processor, and large block transfer through a DMA engine. Both mechanisms incur high overhead: small messages incur an interrupt cost on every message, and large transfers must be initiated by an operating system call. In fact, the DMA facility on the T3D was found to be so expensive (1000 processor cycles to start a transfer) that it was eliminated in the T3E [Sco96].

Alewife [ACD+91, KJA+93, KA93, ACD+95] integrates message passing and cache-coherent shared memory within a single system. Each Alewife node has a hardware controller to handle the common cases of cache coherence, and a DMA unit (in the controller) to facilitate message passing. In addition, the main processor has an efficient memory-mapped interface to the controller that is used for controlling message sends. Though most coherence transactions are handled by the hardware controller, all user messages interrupt the processor for service. Thus, Alewife relies on hardware support for fast processor interrupts. The drawback to this approach is that interrupting the processor can take time away from other computation. In addition, Alewife does not provide support for virtual memory, and provides protection only between the kernel and user processes, leaving user processes unprotected from one another. An extended version of Alewife called FUGU has been proposed to address some of these limitations [MKAK94, MKF+96]. Also, while the Alewife research has addressed the issue of coherence of message data, only local coherence is supported in hardware.

Typhoon [RLW94], a machine architecture proposed by the University of Wisconsin, shares many of the same philosophies as FLASH. The design uses a SPARC processor within the network controller to allow execution of software handlers. Therefore, many of the mechanisms we have discussed for efficiently supporting message protocols are directly applicable to Typhoon, though they have not been studied for that system.

More recently, commercial systems have finally begun to emerge that provide integration of cache coherence and block transfer. Each of these systems has attacked the problem from a different perspective.

Like FLASH, the NUMA-Q system [LC96] from Sequent Computer Systems (previously known as STiNG), also embeds a programmable communication controller within the memory system. In that system, block transfer is not easily supported for a range of reasons, including a more rigid association of (their equivalent of) data buffers and handlers.

The Mercury Interconnect Architecture [WGH+97] was designed by HAL Computer Systems as part of the HAL S1 system. The S1 uses the Mercury Interconnect to join together 4-processor Pentium Pro SMP systems and provide both cache-coherent shared memory and message passing. HAL uses a multilayer approach similar to a traditional network stack to ease the design and verification of the Mercury Interconnect. At the lowest level, the Fast Frame Mover (FFM) provides the analogue to the physical, data link and networking layers of the OSI reference model, concerning itself with the high performance movement of packets. The next level, the Reliable Packet Mover (RPM) provides a reliable transport layer, hiding packet errors occurring in the FFM layer. Finally, the Interconnect Services Manager (ISM) provides high level protocol services such as cache coherence or message passing.

Like FLASH, the S1 system provides a memory copy facility supported by the communication controller. Their conclusions are similar to ours, that the memory copy engine has several advantages: *(i)* it can have more lines outstanding than the processor and *(ii)* transferring directly from source to destination requires only one bus transit per line instead of two [WGH+97]. In their system, the memory copy engine provides significant performance benefits over processor-based copy. In FLASH the benefits from memory copy are more modest, in part because the R10000 processor provides aggressive support for prefetching that allows it to hide memory latency more effectively than the Pentium Pro.

The Silicon Graphics Origin architecture [LL97] and in particular the flagship Origin 2000 system is very similar to FLASH in many respects. It also consists of R10000 processors connected by the CrayLink interconnect. The key difference is that the Origin node controller, called the Hub chip, is hardware-based and not programmable. As a result, Origin is able to achieve slightly lower latencies than FLASH for remote operations. The drawback to a hardware-based approach is that the coherence protocol is fixed and cannot be changed to improve performance, fix problems, or add features.

Origin provides a block transfer engine which provides full integration with cache coherence. The engine is restricted to cache line aligned transfers, making it usable for page copying, but less usable for more general block transfer needs. The block transfer engine is used by the operating system primarily to accelerate page movement and is not exported to the user. Interestingly, Origin overloads directory state bits associated with IO to implement the coherence operations for block transfer. This is similar to our approach in PP ownership, and probably was chosen to reduce the additional protocol support needed to provide this feature.

The S3.mp system at Sun Microsystems is a research project that designed a scalable CC-NUMA machine based on a microcoded controller. Their system provides a number of advanced features that allow the system to be composed of an array of workstations, similar in some respects to Distributed FLASH [KOH$^+$94]. The block transfer facility in S3.mp maintains full coherence at both source and destination. It can send at most 4 KB at a time, with no support for unaligned transfers. Unlike FLASH, which can support many simultaneous transfers, S3.mp provides only one transfer at a time in their microcode. Initiation is somewhat expensive, requiring nearly 10 stores to special I/O registers, and restrictive since it requires physical addresses be provided. The processor detects the completion of transfers through polling, though interrupt support was also planned [Now97].

5.6.1 Discussion

Woo et al. [WSH94, Woo96] studied the benefits of block transfer in scientific and engineering applications, including the impact of prefetching as an alternative to block transfer. Many of their conclusions were similar to ours. They showed that prefetching can achieve similar gains to block transfer in some applications while others performed better using block transfer. They also found that applications are able to effectively utilize the spatial locality afforded by longer cache lines. However, while their study found limited use of block transfer in those applications, the workloads we consider expose some additional opportunities for block transfer, such as in the operating system.

Compared with the aggressive network in FLASH, other systems typically have a larger disparity between local and remote access times. For example, if the network latency increased by a factor of four in the tar workload with faster protocol processing, fbcopy would show a speedup of 1.42 over bcopy, versus 1.22 speedup for prefetched bcopy. In addition, as researchers consider systems in which remote access latencies are even longer, such as the proposed Distributed FLASH machine [KOH$^+$94], coherent block transfer is also attractive for migration and replication of pages to improve memory locality [VDGR96]. The ability of prefetching to hide the long latency of an inter-node block transfer depends on the number of outstanding accesses the processor supports. In our simulations we permit four outstanding prefetches, as in the R10000; more prefetch buffers would decrease the advantages of fbcopy.

Though not provided by the R10000, some modern architectures such as PowerPC and the Sun UltraSPARC include support for block loads and stores. For example, UltraSPARC can load or store blocks of floating point registers directly from or to memory without affecting some levels of the cache hierarchy [Sun95]. This support affords a unique opportunity to avoid cache pollution from processor block copy. However, implementation concerns such as restrictions on the number of simultaneous block operations may limit the applicability of these operations for achieving inter-node transfers.

5.7 Summary

The design of the fbcopy protocol has shown that block transfer can be fully integrated with cache coherence with manageable complexity. Though the performance of a transfer decreases when coherence actions must be taken, fbcopy achieves near-peak network bandwidth when transfer buffers are uncached or cached only locally. Since our fully integrated solution does not degrade performance in locally coherent scenarios, there is little motivation to restrict the implementation to local coherence.

From a performance perspective, though it is not appropriate for all situations, fbcopy offers some advantages over processor copy techniques. Our study of MPI primitives demonstrated that fbcopy enables significant performance gains over implementations of the MPI communication library using processor copy. While the microbenchmark results showed that prefetched bcopy outperforms fbcopy in some cases, through optimizations modeled by faster protocol processing we demonstrated that fbcopy can match or exceed the performance of prefetched bcopy. Fbcopy also results in lower network traffic since it transfers only the data inherently being communicated and avoids polluting the processor's cache with the data moved by the transfer. The results from the tar application also showed that fbcopy achieves competitive performance with a fraction of the CPU occupancy of processor copy techniques. This parallelism can be utilized at the application level to reduce the effective cost of block transfer communication. Finally, while gains from fbcopy in FLASH are modest due to its relatively low remote memory latency, the potential benefits are much greater in systems with longer remote latencies.

Our study of block transfer also illustrates a number of more general conclusions about alternate protocols in FLASH. Particularly significant is the observation that MAGIC can efficiently provide fbcopy using the same hardware support as cache coherence. One of the particular reasons this hardware can support block transfer so effectively is that it uses a generalized data buffer interface. The protocol processor can efficiently allocate and deallocate buffers, and can load and store them with dedicated instructions (`lblock` and `sblock`). This interface is sufficiently flexible that the only change needed for fbcopy is a minor extension to change data alignment. Contrast this approach with other systems that include a programmable controller, such as STiNG [LC96] which associates a single data buffer with a handler. This precludes techniques like chunking that we use to achieve high performance. Overall, this result argues for providing flexible data movement hardware accessed by efficient interfaces, rather than rigid data movement approaches designed for any one protocol.

Chapter 6

FLASH Synchronization Primitives

The second class of communication we study in the context of a programmable communication controller is synchronization. Our study focuses on the two common synchronization primitives described in Section 3.2: locks and barriers. We study conventional implementations of these primitives and attack their shortcomings using support from the embedded flexibility of the FLASH system.

Efficient synchronization primitives are extremely important to support scalable multiprocessors. In his book *High Performance Computer Architecture*, Stone [Sto90] argues that multiprocessor performance is not bounded by aggregate processing power (MIPS or MFLOPS), but that as system size scales, the fundamental concurrency is limited by the frequency with which the system can synchronize. To make the point concrete, he coins the term MSYPS (Million SYnchronizations Per Second) as one possible metric of synchronization throughput. Conventional approaches to synchronization often limit scalability by degrading in performance as systems grow. Our focus in this chapter is to improve synchronization performance (latency and throughput) and maintain these advantages as the machine size grows, in an attempt to increase the scalability the FLASH system can achieve.

Achieving high performance synchronization is a challenge due to its specific requirements. One of the most important features of synchronization primitives is achieving low overhead. Since synchronization primitives are targeted for use within conventional applications using shared memory, the primitives must be integrated with the cache coherence protocol. Our goal of achieving low overhead prompts a different approach to this integration than was used in memory copy. Synchronization also entails the interaction of many different processors, unlike memory copy which (in the common case) involved two processors in tight communication. This characteristic requires our protocols to take special steps to avoid contention by distributing work around the system.

This chapter is organized as follows: In Sections 6.1 and 6.2, we present the lock and barrier protocols in detail by describing:

- the unique performance metrics for the synchronization primitive.
- the benefits and shortcomings of existing techniques and the opportunities to improve upon them using custom protocol support.
- the design and implementation of a custom FLASH version of that primitive that achieves these goals.

In each section, we consider the performance of the primitive in isolation, similar to the analysis of FLASH memory copy in Section 5.4. By studying each primitive in a controlled environment and in the context of microbenchmarks, we can more clearly identify its performance characteristics and its potential to improve real application performance. In Section 6.3 we consider the performance impact of these primitives in the context of scientific applications taken from the SPLASH benchmark suite.

6.1 FLASH Locks

The first synchronization primitive we consider as a candidate for custom FLASH protocol support is locks. A lock provides *mutual exclusion*, i.e., only a single processor can hold a lock at any one time. Locks are typically used to assure exclusive access to critical resources, code, or data structures. A processor requests the lock through a *lock* operation, and releases it with an *unlock*; the time during which the lock is held is known as a *critical section*.

Our motivation for considering locks for custom protocol support stems from some fundamental observations about the performance of locks implemented on top of shared memory. On the one hand, as we discussed in Section 3.2.1, shared memory locks tend to perform fairly well in situations where contention is low. There has been considerable focus on improving shared memory locking performance: the cache coherence protocol is well-tuned for simple line exchanges, and the caching of locks enables rapid re-acquisition by the previous holder. However, shared memory locks degrade significantly in performance and characteristics as contention increases.

We begin in Section 6.1.1 by describing the desirable characteristics of locks overall. Then, to motivate the need for protocol support, we focus in Section 6.1.2 on the drawbacks of a few representative conventional implementations of locks and examine in detail the source of performance degradation under contention. In Section 6.1.3 we describe the application interface to FLASH locks.

Sections 6.1.4–6.1.7 describe the detailed implementation of the lock protocol, highlighting how it attacks some of the shortcomings of conventional locks through specialized protocol support. In Section 6.1.8 we study the performance of FLASH locks in isolation. We present application results using these locks later in the chapter (Section 6.3) after presenting the protocol for FLASH barriers. Finally, Section 6.1.9 considers the broader related work on locking synchronization primitives.

6.1.1 Metrics for Evaluating Lock Performance

We begin by our discussion of locks by describing the desirable characteristics of a lock primitive. We explore both qualitative features that are desirable as well as quantitative performance metrics used to compare lock implementations cleanly. We refer back to these metrics later to evaluate the success of our lock implementation.

Fairness

One important characteristic of locks is providing *fairness* for all lock accesses. In other words, given simultaneous requests for a lock each processor should be equally likely to succeed in acquiring it. Furthermore, if a processor does not succeed, it should effectively have higher priority in successive attempts (providing a round-robin behavior). Among other drawbacks, a lack of fairness can decrease performance by introducing a load imbalance in the system. This can arise in several different ways. For example, if a lock were used to dynamically distribute work to processors, a lack of fairness would cause some processors to wait longer to allocate work and thus waste otherwise useful time. Alternatively, if work is distributed statically, a lack of fairness in access to shared resources can cause some processors' work to take longer than others, also causing a load imbalance.

A severe breakdown in fairness may result in a condition called *starvation* in which a processor never succeeds in acquiring the lock and thus cannot make forward progress. Research in the FLASH project found that when nodes are connected by a low-dimensional network, processors which are far from a memory location can easily be starved due to network effects in combination with the cache coherence protocol [Hei97]. Consider the example of many processors attempting to gain ownership of a lock line. Under network congestion that causes negative acknowledgements, nearby nodes can retry very quickly while a far away node takes longer and may never win the race to the home. The nature of this problem is such that it may not be possible to assure fairness through processor software techniques alone.

Low Overhead / High Throughput

In general, the goal of parallel applications is to scale in performance as the machine size grows. Of course, one aspect of scalability comes from the application, which must be structured to remove real serialization bottlenecks. But a second critical aspect is to provide high synchronization *throughput*, which in this case corresponds to locks being acquired and released at *low overhead*. Overhead in locks corresponds to time spent during the acquisition of an available lock or the transition of the lock from one holder to the next. In each case, the dead time due to synchronization overhead renders the lock-protected resource unavailable to the application even though it is actually unused.

The most difficult challenge in achieving low overhead in locks is assuring that the overhead remains low as the machine scales and contention increases. Unfortunately, synchronization primitives constructed on top of cache-coherent shared memory often do not match the inherent communication needed to synchronize, but instead incur communication which is an artifact of the coherence protocol. Such artifactual communication can worsen as more processors attempt to acquire the lock, causing performance to degrade. We study this problem in detail in the next section.

For the purposes of measuring locking overhead, we consider two different classes of accesses to locks: contended and non-contended. We separate these classes since they isolate different features of the lock implementations we study. In each case, we isolate a performance metric which reflects the latency which is the locking bottleneck.

Non-contended Acquire Latency

We begin with non-contended locks, ones which are found to be free when requested. In this case, the latency to acquire the lock is set by the duration of the lock primitive itself, which we refer to as *acquire latency*. Figure 6.1 illustrates this latency in several situations, showing in particular that acquire latency is not necessarily constant. In the first acquisition by node B, the lock was previously held on node A and so the lock must be fetched remotely. In the second acquisition, node B re-acquires the same lock again, which may be faster in primitives that support caching of locks.

Furthermore, locking performance may also vary based on *which* nodes acquire the lock, since requests for memory are typically sent to the memory's home. Therefore, lock requests by the home node or requests for locks whose previous holder was the home have higher performance than exchanges between remote nodes in some lock implementations.

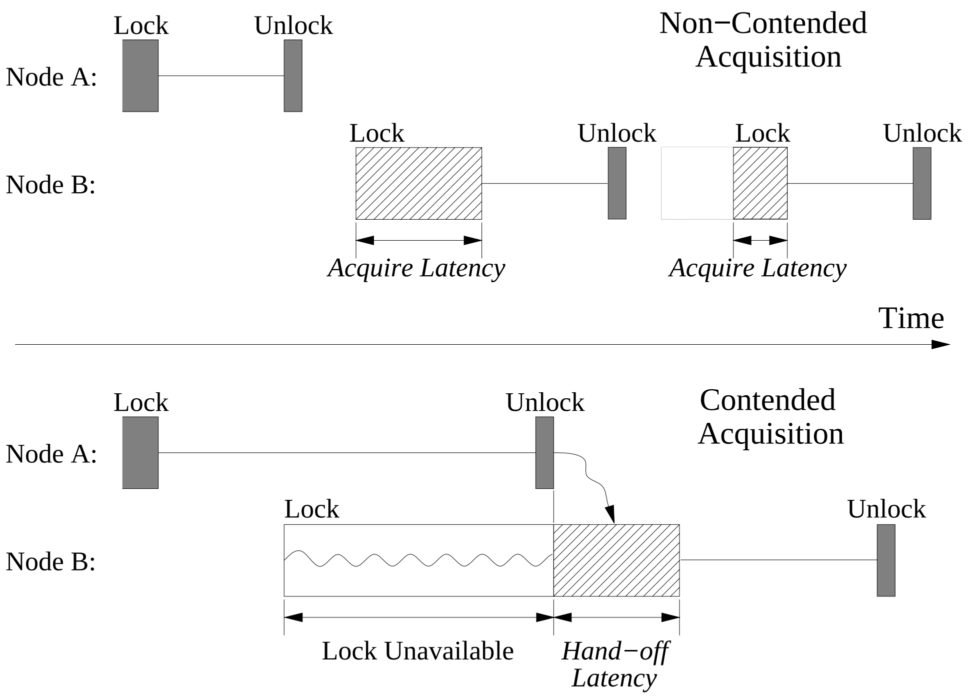


Figure 6.1: Non-contended and contended lock acquisition latencies. Non-contended acquisition *acquire latency* is measured from the time the lock primitive begins until it completes with the lock (*top*). Contended acquisition *hand-off latency* is measured from when node A releases the lock until the lock operation on node B completes, since the duration the lock is held by node A is function of the application not the lock primitive (*bottom*).

Contended Hand-Off Latency

We separately consider the performance of contended locks, ones which are found to be held by another processor when requested. In this case, we cannot use the acquire latency metric to evaluate the lock, since the lock is unavailable when requested and the waiting duration is dependent on the application and not the locking primitive. Instead, the critical metric for contended locks is the time taken to hand-off the lock *once it is released*. This metric, called *hand-off latency*, ignores the time spent waiting for a busy lock and focuses on the remaining overhead during exchanges. Figure 6.1 also illustrates hand-off latency. As illustrated, we measure hand-off latency as the duration from when the unlock primitive completes (indicating that the lock was released) until the new holder's lock primitive completes (indicating that it has successfully acquired the lock).

6.1.2 Conventional Lock Implementations

We begin by studying conventional lock implementations to understand their advantages and shortcomings. One drawback we study in particular is performance degradation under contention. Our simulation results from controlled stand-alone tests in the absence of contention show that a basic shared memory lock can be acquired from a remote node in about 1 μs. When contention is introduced—128 processors concurrently accessing the same lock—acquiring the lock takes 57 μs on average. In our application results, which include contention from normal shared memory accesses as well, shared memory lock acquisition latency can exceed *several hundred* microseconds. We analyze two implementations below to understand the source of this slowdown.

LL/SC-Based Lock

Conventional lock implementations utilize the atomic memory update primitives provided by the processor to modify a shared memory location. In the R10000 (and other processors such as the DEC Alpha), atomic memory updates are provided by a pair of special memory operations called *load linked* and *store conditional* (commonly referred to as *LL/SC*) [MIP96, SW95]. Load linked operates as a normal load, except it causes the processor to watch for other transactions on the address which was read. The processor can then update the data in its cache as desired. Later, the store conditional to that address only succeeds if no conflicting operations occurred for that line since the LL, and the line is still intact in the cache. If a conflict does arise, such as another store occurred since the LL executed, the SC does not modify the memory, but instead returns a failure result code in a register. This failure can be detected and the entire LL/SC sequence can then be retried until the sequence completes atomically [Gha95, MIP96]. We use exponential back-off between failed lock attempts to reduce subsequent contention.

A basic lock implementation, which we refer to as LL/SC locks, uses the atomicity provided by LL/SC atomically updated a shared flag. We present its implementation in Appendix B, Figure B.1. The direct usage of these primitives is very fast in the absence of contention, and allows locks to be cached and thus reacquired rapidly. However, as contention increases, the performance of the LL/SC lock rapidly degrades.

The inefficiency of LL/SC locks due to contention arises from the characteristics of the cache coherence protocol. A lock represents a unique entity that only one processor can hold at a time. Cache coherence, on the other hand, is fundamentally designed to permit the seamless *replication* of data. Consider the situation illustrated in Figure 6.2, in which one processor (node 0) is holding the lock and several others (nodes 1–4) are waiting for it to be released. Node 0 finishes its use of the lock and wants to release it. Since nodes 1–4 are spinning on the lock address, node 0 no longer has exclusive ownership and must request an upgrade, which invalidate the sharers. In Figure 6.3, the lock holder is granted ownership and can release the lock, as the former sharers acknowledge the invalidations. Following the invalidation, the requesters all miss in their cache in quick succession, and request the lock. This causes a rush of requests to the home node, illustrated in Figure 6.4. The first request to arrive is sent to Node 0 to fetch the modified copy from the cache, while the other

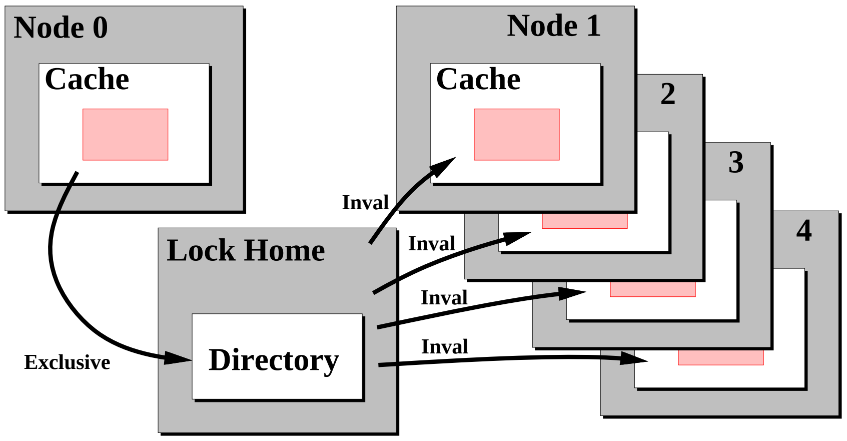


Figure 6.2: Invalidation of lock waiters when the holder upgrades to release the lock.

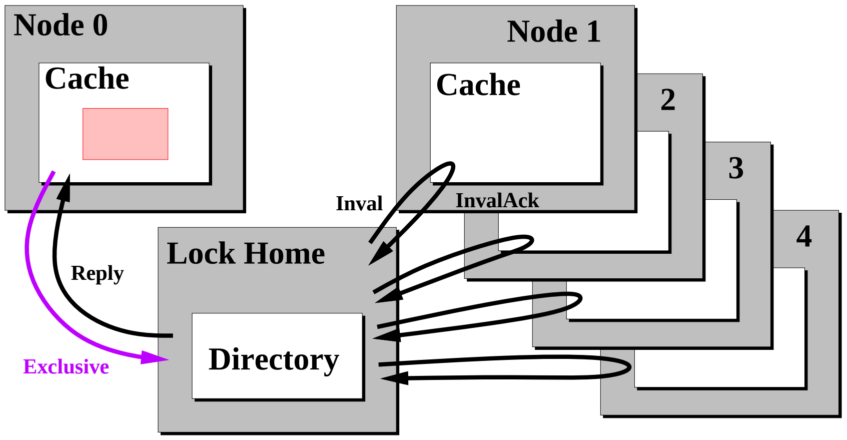


Figure 6.3: Former sharers acknowledge the invalidation. The lock holder gets exclusive access and can release the lock.

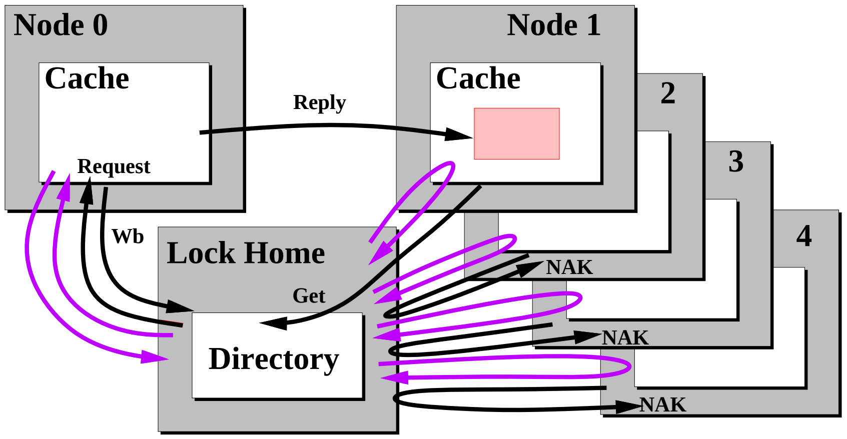


Figure 6.4: A rush of requesters follows an unlock. One requester gets the lock line first while others are negatively acknowledged. Note the accumulated traffic from this exchange is significant, though little communication inherently occurs.

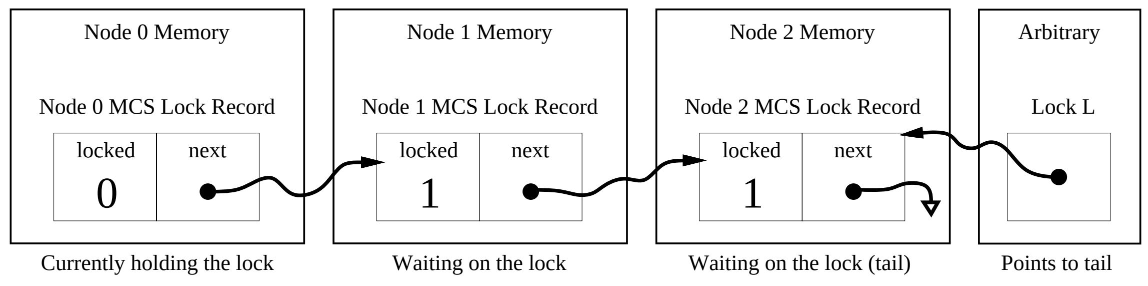


Figure 6.5: MCS Lock queueing approach. MCS Locks consist of an array of lock structures, connected in a linked list fashion to form a queue. Each node spins on the lock bit in its record, allowing waiters to be released individually. The lock pointer L indicates the tail of the list, where new waiters are added.

nodes receive negative acknowledgements because the line is marked pending. The rush of requests swamps the protocol processor, preventing the lock writeback from occurring and delaying other requests for that node.

Furthermore, since the decision of who holds the lock next is purely based on a race to the home, this implementation also fails to provide fairness for the ordering of lock acquisitions. In fact, starvation or livelock may result if certain processors tend to consistently lose the race for the lock, for example because of network topology, or if where ownership is granted and then stolen away before the lock can be acquired [KCA92].

To address both the problem of fairness and the rush of requesters, ideally the lock would be provided to the next holder more directly. In fact, some coherence protocols have proposed extensions to the base protocol to improve transactions such as lock hand-off [CBZ91, BZS93, KCDZ94, KBG97]. The other lock primitives we study provide this feature through queueing of requesters.

MCS Lock

As the previous section shows, LL/SC suffers from serious problems due to contention. One way to address the contention problems of LL/SC locks is to implement a *queue* of lock waiters in *software*. One such implementation is called MCS Locks, proposed by Mellor-Crummey and Scott [MCS91a]. For systems without hardware support for locking, MCS locks are one of the most efficient primitives [MLH94, KBG97]. Many other implementations of software queued locks have also been proposed, each with slightly different characteristics; we consider those other implementations in Section 6.1.9.

The pseudo code for the MCS lock implementation used in our study is shown in Appendix B, Figure B.2. An MCS lock is implemented as a linked list of lock structures, illustrated in Figure 6.5. Each structure corresponds to a waiter for the lock; the head of the list is the current holder. By

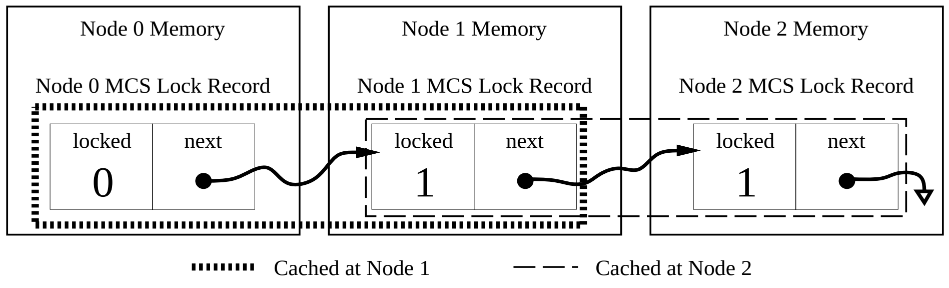


Figure 6.6: Extraneous caching in MCS lock queues. The process of enqueueing a record through shared memory operations caches the previous node's record as well. The extra shared copy must be invalidated when the lock is granted to that node. For example, Node 1's MCS lock record is cached by node 2. When node 0 clears node 1's lock flag allowing it to continue, an invalidation is generated to node 2 as well.

making each waiter spin in its own (locally allocated) structure, MCS locks prevent any one node from becoming a hot spot.

Since MCS is a queued lock, it is able to provide fairness and significantly reduce the contention-based performance degradation which occurs in LL/SC-based locks. MCS locks incur no rush of requesters at an unlock. Instead, the next waiter is individually released from spinning to acquire the lock. However, since MCS locks are built on top of shared memory, they still encounter two sources of artifactual communication.

First, the MCS queue tail is updated internally with LL/SC to assure atomicity. As a result, contention can arise from simultaneous requests for the lock. Fortunately, unlike LL/SC locks, contention of this nature only arises on the *initial* request for an MCS lock. Once a requester is queued for the lock it spins locally in its cache until its turn arrives, avoiding further contention.

The second source of artifacts arises when a lock waiter is released. As explained above, only the next waiter is released on an unlock. However, the construction of the queue in software causes extra caching of MCS lock records, as illustrated in Figure 6.6. The source of this caching is as follows: when a node adds itself to the queue through shared memory accesses, it caches the previous node's record to update its `next` pointer. The previous node is still spinning on the `locked` bit, so it re-requests the line, leaving a shared copy at both nodes. As a result, releasing the next waiter requires an invalidation be sent to *its successor node as well,* which is purely artifactual.

6.1.3 FLASH Lock Application Programming Interface

The goal of our lock implementation is to use the support provided by the programmable communication controller to further improve locking performance. Of course, our approach is merely one possible implementation of locking using the flexibility in FLASH. Just as in FLASH memory copy, we export lock functionality to the application through a library call. This allows us to hide the details of the locking implementation and export a simple array of numbered locks to the system. We provide two basic locking calls:

```
void FLASHLock (int lockNumber);
void FLASHUnlock (int lockNumber);
```

`FLASHLock` requests that the numbered lock be acquired for the process. This call spins internally until the lock is held. `FLASHUnlock` releases a lock held by the application, passing it on if another process is waiting.

A third call, the implementation of which we leave to the operating system designers, is responsible for allocating a physical range of locks to an application that requests it. The allocation of locks is analogous to physical memory management. Internally, FLASH "physical" lock numbers are unique across the entire machine. By mapping each application to use different physical locks, the operating system can virtualize this resource like physical memory and allow each application's "virtual" lock range to begin at zero.

Goals of FLASH Locks

The primary goal of FLASH locks is to address the limitations associated with conventional shared memory locking implementations. In particular, as we showed earlier, shared memory locks may incur significant artifactual communication under contention. Our lock primitive specifically targets these high contention situations, and strives to eliminate artifactual communication *completely*. We pay particular attention to optimizing the lock hand-off following an unlock. A second goal is to assure fairness across lock requesters and eliminate the rush of requesters following an unlock. Our protocol uses queueing to help achieve both of these goals.

We also attempt to match the advantages of shared memory locks in non-contended situations, such as low latency to acquire a free lock. In addition, we aim to support caching of locks to permit efficient repeated acquires by the same processor, another worthwhile characteristic of shared memory locks.

6.1.4 FLASH Lock Implementation

This section describes the implementation of the FLASH lock protocol in detail, focusing on its differences from the conventional cache coherence protocol and the approach used in FLASH memory copy. We begin by presenting the protocol operation itself, describe its protocol state layout, and explain the unique way we support cached lock state. In Section 6.1.5 we discuss several key design choices and the alternatives approaches we did not choose. Section 6.1.6 explores two subtle issues that arise from multiprogramming and from processors failing to acquire a lock in a timely fashion. That section describes techniques to assure the protocol is robust in the face of these situations.

Protocol Overview

At a high level, the operation of lock protocol is somewhat similar to shared memory locks based on exclusive ownership of cache lines. As in shared memory, locks are represented in the cache and are acquired using atomic memory update primitives. Also like shared memory locks, a cached lock can be acquired and released repeatedly without external communication (assuming another processor has not requested it). The major differences are in the underlying protocol implementation and the exact interface to request and release locks. These differences allow the protocol to match the inherent communication needed for locking and avoid the drawbacks of shared memory locks.

The most significant change as compared to the cache coherence protocol is that the lock protocol stores lock state in MAGIC at each node. This enables the protocol to construct a distributed queue of waiters that can pass the lock directly to the next holder in a *single message*. As in cache coherence, each lock is assigned a home node, but in this protocol the home's only responsibility is tracking the tail of the queue of waiters.

Internally, the protocol uses a concept called a *token*, which is analogous to ownership of a cache line, but has different properties that are more compatible with locking. In short, tokens represent permission to acquire a lock, similar to the way a token-ring networking protocol allows only one sender at a time. Unlike ownership, however, which may be stolen away while a lock is held (Figure 6.2), tokens must remain at the node while the lock is held. We explain below how tokens are manipulated and identify the advantages of this approach.

Lock State

First we describe the lock protocol state, to illustrate how the protocol stores locks internally. Figure 6.7 illustrates the state, which consists of 64 bits per lock stored at each node. The fields are used as follows:

Token (T) Indicates whether the token is present. The processor spins on this bit when it waits for the token to arrive, and clears it when it yields the token.

Locked (L) Indicates whether the lock is currently locked. If the processor finds the token present, it acquires the lock by setting this bit. Just as in any read-modify-write operation, there is a race in which the processor sees the token present, but then loses the token before the Locked bit is set, so LL/SC is used to assure atomicity when setting this bit. This does not suffer the same problems as LL/SC locks because this state is local to the node.

Seen (S) Indicates if the processor has acquired the lock at least once after being granted the token. Our protocol guarantees the lock can be acquired once, to assure fairness and eliminate the "window of vulnerability" problem in conventional locks [KCA92]. To provide this, the Seen

T	L	S	E	C	H	R	I	Q	Next Holder	Home/Queue Tail	Queue Index	unused
1	1	1	1	1	1	1	1	1	12	12	8	23

Figure 6.7: Lock protocol internal state format.

bit is cleared when the token is granted and an arriving token request is queued if it finds Seen is still clear (even if the lock is unlocked). In the same atomic sequence that sets the Locked bit, the processor sets the Seen bit as well. Though it is critical to fairness, the Seen bit has some subtle correctness implications. In particular, the lock *must* be acquired after the token arrives at the node or Seen prevents it from being claimed by another node. We expand on this issue later and describe extensions to the protocol to ensure correctness in all cases.

Externally Requested (E) Indicates whether other processors have requested the token. The node retains the token at an unlock if this bit is cleared, otherwise it asks MAGIC to send the token to the next processor in line.

Cached (C) Indicates if the processor is currently caching the lock state. This tells the protocol if the processor cache must be consulted to read the current Locked bit value.

Home (H) Indicates if this node is the home for this lock. Unlike shared memory locks such as those in DASH, since each node stores lock state any one can be used as the home for the lock. The application designates a home node for each lock when it is initialized, allowing the locks to be spread across the machine easily.

Requested (R) Indicates whether the processor has requested the token for itself. It is cleared when the token is granted. We describe in Section 6.1.6 how this bit is needed to solve a locking problem that arises in multithreaded environments.

Initialized (I) Indicates the lock has been configured and is ready for use. Access to an uninitialized lock should deliver an exception to the application via the operating system.

Queue (Q) Indicates that the NAK avoidance queue is currently in use. The lock protocol is able to avoid negative acknowledgments in almost every case because processor requests guarantee outgoing queue space. The one exception is when the Home node forwards a token request to the queue tail (described below) in which case, just as in cache coherence, the incoming request is not guaranteed outgoing request space. If the outgoing queue is full, we could NAK the original request, however this significantly complicates the protocol and reintroduces many of the problems of conventional locks. Instead, we maintain a simple circular queue at the lock home (not shown) to defer requesters *in order*, thus eliminating NAKs completely. Deferred requests are completed using the software queue when queue space becomes available. This approach brings the protocol into full compliance with FLASH deadlock avoidance conventions and assures fairness in every situation without impacting common case performance.

Next Holder Used to store the successor node, the one which will be granted the token when this node unlocks.

Home/Queue Tail For a node which is not the home, this field indicates which node has been designated the home for this lock. The home uses this field instead to indicate the current queue tail of the lock.

Queue Index Since lock homes are assigned dynamically, the lock allocates a NAK avoidance queue from a preallocated pool when it is designated the home and stores the queue index number in this field.

Though it incurs higher state overhead than cache coherence, maintaining protocol state at each node has several particular advantages. First, it enables the protocol to construct a distributed queue that enables very efficient hand-off. It also enables MAGIC to store the token reliably when it arrives and save it locally if replaced from the processor cache (instead of sending it to the home). The former use is critical to allow the protocol to *push* a token to a node (similar in many respect to an update-based coherence protocol), a feature otherwise impossible in FLASH because the processor does not support update operations into the processor cache itself.

A similar technique was used in DASH for its cache coherence implementation, using a board-level cache called a *Remote Access Cache* (RAC). Two of the major reasons to use a RAC were to hold *(i)* lines supplied from a remote node before a local processor cache completes its read request (an artifact of the DASH implementation, which required the read to retry after the line arrived at the node), and *(ii)* remote lines replaced by the local processor cache [Len92].

Lock Protocol Operations

Building on the protocol state explanation, this section describes the operation of the lock protocol by illustrating how the protocol handles the different locking situations that can arise. Unlike the cache coherence protocol which is extremely complicated, the diagrams illustrated here show essentially the entire protocol. This concise design is one of the benefits of our implementation, allowing it to share MAGIC caches with the cache coherence protocol more effectively than a heavier protocol like memory copy. Figure 6.8 describes the symbols used in the diagrams.

To acquire a lock, the processor begins by reading the lock address in its cache to see if it holds the token. The result is not cache-coherent memory, but a copy of the lock state from the local MAGIC chip, as illustrated in Figure 6.9. If the token is present, a conventional LL/SC sequence is used to atomically set the Locked bit, just as in shared memory locks.

Ⓣ	**Token Holder**	The node holding the lock token.
Ⓡ	**Requester**	A node requesting the token.
Ⓠ	**Queued Requester**	A prior requester, queued for the token
Ⓗ	**Home**	The lock's designated home node.
➡	**FLASH Message**	A message between two MAGIC chips (or between MAGIC and the processor cache).
→	**Node Pointer**	Internal protocol state that points to another node (no communication is implied).

Figure 6.8: Legend of symbols in lock protocol diagrams.

Unlike shared memory locks which acquire ownership automatically upon a write, if the token is not present the node must request it explicitly using an uncached command to MAGIC. Here a single write (a PPR) is sufficient to express the token request. In Section 6.1.5 we explain how an explicit request is needed due to the combination of processor speculation and the fairness guarantee we provide. The PPR causes MAGIC to request the token from the lock's home node, as illustrated in Figure 6.10.

The home's only responsibility in the lock protocol is to track the last node which requested the lock, the *queue tail*. When it receives a token request, it forwards it to the queue tail and then updates the tail to reflect the new requester. At this point, three cases may occur:

No queue exists, the lock is unlocked. Since no queue exists, the forwarded token request reaches the token holder, illustrated in Figure 6.10. There, MAGIC extracts the cached lock state from the processor to see if the Locked bit is set. In this case it finds it unlocked, so it forwards the token to the original requester. Unlike cache coherence, no acknowledgment is sent to the home. When the token reaches the requesting node, MAGIC invalidates the cached state on which the processor is spinning. The processor re-reads the state and, seeing the token, acquires the lock by setting the Locked bit as before.

No queue exists, the lock is locked. In this case, illustrated in Figure 6.11, the token holder MAGIC finds instead that the processor has currently asserted the lock. It is thus unable to yield the token, and instead stores the requester's node number as its successor. By indicating that the lock has been requested, the processor will yield the token when it unlocks, as described below.

- Read of lock state misses in processor cache (1).
- MAGIC generates cached representation from internal lock protocol state, noting it is cached by the processor.
- Reply to processor with lock state (2),
- If token is present, processor atomically sets the Locked bit in its cache, asserting the lock.

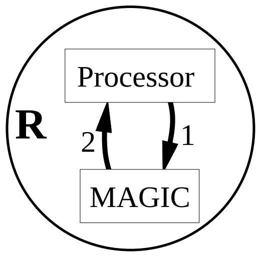


Figure 6.9: Request for local lock state from MAGIC.

- Processor reads lock state (Figure 6.9) and finds token absent.
- Processor issues token request PPR to MAGIC, causing request to lock home (1), then spins on the lock state waiting for the token.
- Home forwards request to queue tail (2), internally points to node R as new queue tail (a).
- Token holder MAGIC extracts lock state from processor if cached. Finding it unlocked, it forwards the token to node R (3).
- At node R, MAGIC invalidates cached lock state, causing spinning processor to re-request it.
- With the token now present, Node R sets Locked bit in its cache, asserting the lock.

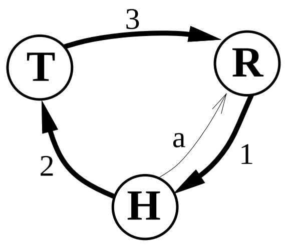


Figure 6.10: Token request for lock currently at another node, no queue pending (Unlocked).

- Node R requests token from home as before.
- Home forwards request queue tail (2), internally points to node R as new queue tail (a).
- Token holder MAGIC extracts lock state from processor if cached. Finding it locked, it internally points to node R to receive the token next (b), forming a queue.
- It also marks lock as requested to indicate the processor should yield the token on an unlock (Figure 6.14).

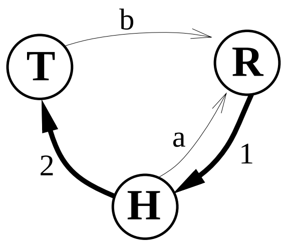


Figure 6.11: Token request for a lock currently at another node, no queue pending (Locked).

- Node R requests token from home as before.
- Home forwards request to queue tail (2), points to node R as new queue tail (a).
- Queued node internally points to node R as its successor (b).

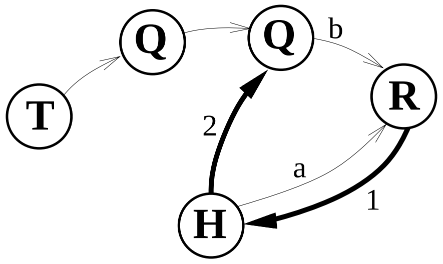


Figure 6.12: Request for a lock with a queue of waiters pending.

- Unlock operation clears Locked bit, requesting lock state from MAGIC if not cached.
- Since state indicates no external requesters, the token is kept, the lock state remains cached, and no external communication is required.

Figure 6.13: Unlock operation in cache when no queue is pending.

- Unlock operation clears Locked bit, requesting lock state from MAGIC if not cached.
- Since state indicates a queue is present, processor issues unlock PPR to MAGIC and clears token bit.
- MAGIC sends token to successor node (1).
- Successor MAGIC supplies token to processor as in Figure 6.10.
- Note that lock Home is not involved in lock hand-off.

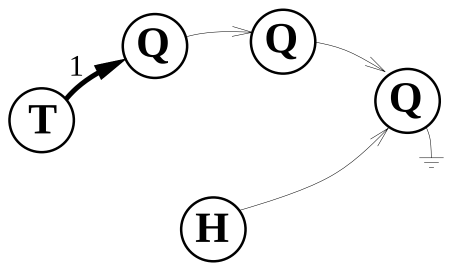


Figure 6.14: Unlock operation with one or more requesters pending.

A queue exists for the lock. In this case, the home node's queue tail points not to the token holder, but to the last node in the queue of waiters. When it receives the forwarded token request, the current queue tail stores the requester's node number as its successor, illustrated in Figure 6.12.

Now we turn to the unlock operation, which is extremely efficient in this protocol. To unlock, the processor first clears the Locked bit in the cached lock state, fetching it from MAGIC if it is not cached.[1] Then it consults the state to see if another node has requested the lock. In the first case, no request has been posted, so it retains the token and the lock state remains in its cache (Figure 6.13). Subsequent lock operations (Figure 6.9) can succeed in the cache without interaction with MAGIC if no other intervening request occurs.

On the other hand, the unlock may see that another node has requested the lock, illustrated in Figure 6.14. To assure fairness, our protocol requires the token be yielded; the processor yields the token using a second kind of PPR to reliably indicate to MAGIC it has unlocked. MAGIC clears the token indication and sends it to the successor node it stored previously. When the token reaches the new holder, it is provided to the processor as usual, by invalidating the cached lock state. Note that this hand-off consists of a single message and does not involve the home.

Efficient Lock Caching

As we described, our protocol differs from cache coherence in that cached lock state is not global memory, but is a representation of the local MAGIC's lock protocol state. MAGIC constructs the

[1]A request may have extracted the line to consult the Locked bit, or the state may be replaced from the cache due to a conflict, just as any other line.

processor's representation by reading the lock protocol state from the PP cache and storing it in a data buffer explicitly. Rather than add special cases to the coherence protocol's miss handlers to provide this feature, we use FLASH's address space capability (explained in Section 4.1.1) to indicate locks addresses are different than conventional memory. We program the Jump Table to select a miniature coherence protocol (part of the lock protocol) to service cache misses in this "lock space". We use a similar technique in our implementation of FLASH barriers in Section 6.2.

It may seem surprising to use a separate coherence protocol just for lock requests, but in contrast with the sizeable cache coherence and memory copy protocols, the entire lock protocol (including the coherence handling) consists of only *seven handlers*, occupying less than 3 KB. This concise design is possible because the lock coherence protocol is not required to handle the huge number of cases that arise in a normal cache coherence protocol from the interaction of local and remote caching. Instead, by separating requests for the current lock state (intra-node communication) from requests for token movement (inter-node communication), the number of cases can be dramatically reduced. In Section 6.1.7 we explore the specific components of the protocol to provide an overview of how it is structured.

6.1.5 Protocol Discussion

The lock implementation we describe is only one possible implementation we might use. In this section we briefly discuss two design issues for the protocol to illustrate why several features were made. First we consider a different queueing approach and illustrate its drawbacks. Then we discuss the impact of processor speculation on the protocol design.

Centralized Queued Locking

Our initial implementation used a centralized queue of waiters, stored at the lock home. In that approach, requests for the lock token are sent to the home followed by a later reply from home with the token. Unfortunately, the home is a serious bottleneck since it is involved twice for each contended lock acquisition: *(i)* when the request arrives for the token, causing the requester to be queued, and *(ii)* when the token is released by the previous holder in the queue. Furthermore, the token hand-off is slower since it must traverse through the home to determine which node should receive it next.

We also considered a modified version that targets this inefficiency by telling the current holder in advance who the following holder will be. This implementation, illustrated in Figure 6.15, allows the holder to forward the token, then notify the home node outside the critical path. In response, the home issues a new forwarding request to the new holder. Though this optimization improves the lock hand-off latency significantly, the home is still consulted twice, so for short critical sections

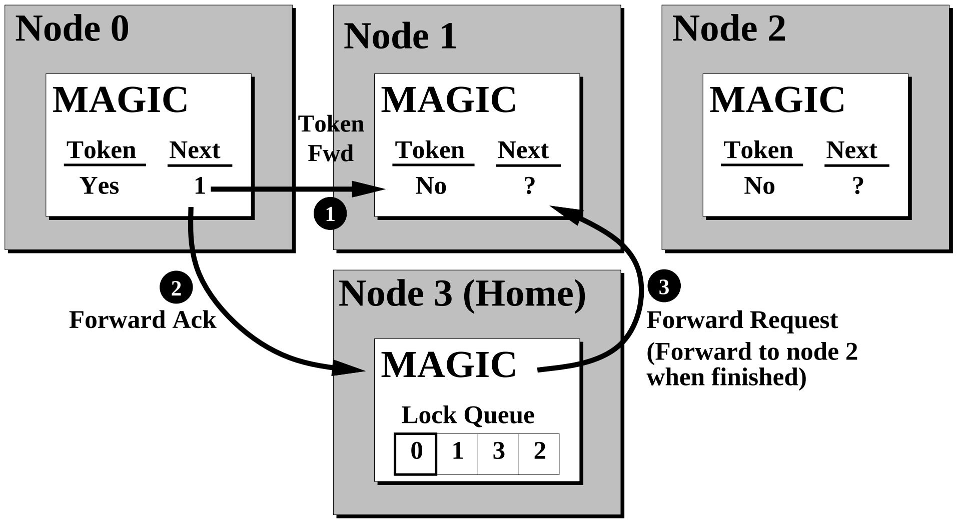


Figure 6.15: The centralized queue locking approach for FLASH locks. The queue is stored completely at the home; only the current token holder knows which node follows it in the queue. The arrows illustrate the token exchange process, which both passes the token to the next holder and sets up the next round of forwarding.

it limits performance. The distributed queue approach we use provides the same direct forwarding benefits of this centralized queue, but further improves performance by completely eliminating interaction of the home after the initial request.

Avoiding Problems from Processor Speculation

Recall that token requests are explicit in the protocol and not associated with a lock state request. It may seem like a natural optimization to request the token whenever the processor fetches the lock state to launch the request as early as possible. Unfortunately, the aggressive speculation performed by the R10000 combined with the fairness guarantee of our protocol makes this approach unusable. If the protocol were to request the token because of a speculative request and then the processor never actually acquired the lock, the cleared Seen bit would prevent the token from leaving the node and deadlock would occur.[2] As a result, token movement must only occur coincident with a definite intention to acquire the lock, so we use uncached operations which are never speculated. As we show in the microbenchmark analysis, the inability to launch the token request immediately increases FLASH lock non-contended acquire time as compared to shared memory approaches.

Similarly, the protocol carefully defines the semantics of cached lock state such that the presence (or absence) of the lock state in the processor's cache does not by itself indicate whether the lock is held. Lock state in the cache merely indicates that the processor may be *interested* in the lock. This

[2]In fact, the timeout implementation described in Section 6.1.6 eventually reclaims the lock to avoid deadlock, but the situation should be avoided where possible.

is essential because the processor may speculatively request the lock state, and conversely because cache conflicts may cause the lock state to be replaced while it is still in use. Instead, the *value* of the Locked bit (which is never speculatively modified) is used to indicate its status, similar to shared memory locks.

6.1.6 Multiprogramming/Multithreading Issues

Despite the careful effort to prevent token requests from ever occurring speculatively, there are still some situations in which a token that arrives at a node is not subsequently acquired. One such situation occurs in multiprogrammed or multithreaded environments where a processor requests a token and then is somehow preempted before it arrives. The process may also migrate to another processor before it resumes. A related problem occurs when two threads (or processes) executing on the same node try to access the same lock. This section describes solutions to these problems. For the remainder of this section, we use “multiprogramming” and “process” terminology for brevity, but identical issues arise from multithreading and our solutions apply there as well.

Reclaiming Unused Lock Tokens with Timeouts

Previously, we assumed that a token could only be requested by an application which was actually intended to acquire it. Unfortunately, multiprogramming may preempt such a process before the requested token arrives. Depending on the situation, the process might be descheduled for a significant length of time during which the cleared Seen bit prevents other nodes from taking the token.

If the preemption is long enough, it may be in the best interests of overall performance to reclaim the token to allow other nodes to use the lock in the interim. Similarly, if the process is stopped (or worse, killed), then correctness requires that the token be reclaimed to allow other processors to use it. Note that in both cases we are referring to a lock which was not actually acquired; if the node acquired the lock then it must be retained to assure correctness of the critical section. If a process dies or is suspended while holding a lock, operating system intervention is required to restore the system to a consistent state.

We handle both of these cases through a software-managed timeout mechanism. The effect of this mechanism is to change the Seen bit guarantee (described previously): the processor is still assured the ability to acquire the lock at least once, *but only within a certain timeout region*.

The difficulty in providing this mechanism is implementing it efficiently. Different applications could be using many locks simultaneously on the same processor, each of which requires its own timeout. There are many approaches that might be used to address this problem. A naïve approach might store a “token grant time” in each lock record and then periodically walk the entire array of locks, checking to see if any locks have exceeded the allowed timeout. The performance of

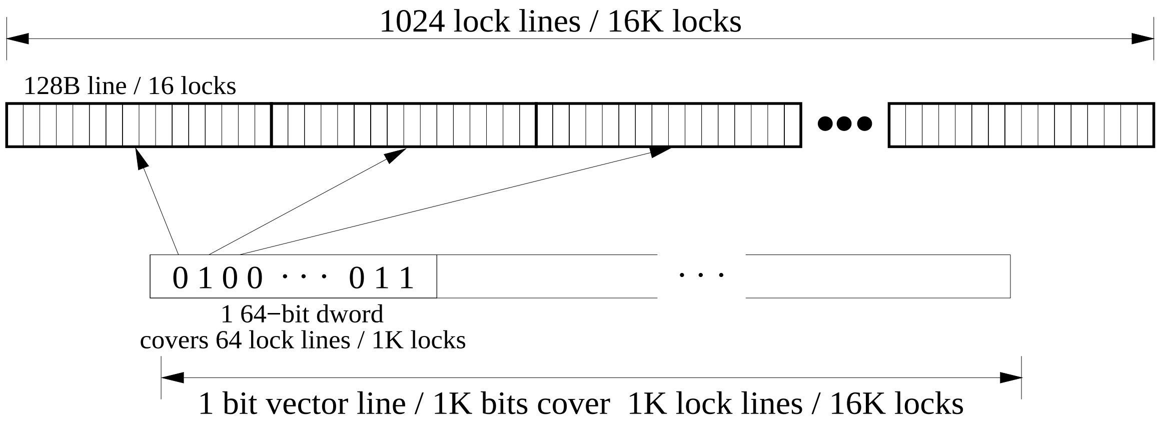

1024 lock lines / 16K locks
128B line / 16 locks
0 1 0 0 · · · 0 1 1
1 64−bit dword
covers 64 lock lines / 1K locks
1 bit vector line / 1K bits cover 1K lock lines / 16K locks

Figure 6.16: Timeout bit vector structure and lock correspondence.

this approach is incredibly poor since not only must it examine each lock record, but it also incurs PP data cache misses for the locks which were not actually used. This cache traffic makes this approach prohibitively expensive and seriously disrupts the cache working state. On the other end of the spectrum, we might maintain a linked list of locks that were recently granted tokens and their arrival times, and periodically walk the list. For very light lock usage, this approach may be ideal, but with even mild usage the timeout data structure can grow to a significant size and experience similar cache drawbacks.

Instead, the approach we choose attempts to balance these two extremes and eliminate extraneous cache traffic associated with timeout checking. It is based on a bit vector indicating which locks have received token grants recently. Locks indicated by the bit vector are scanned periodically to verify they have not timed out since the token grant. The key innovation in this approach is that it marks recent token grants at a carefully selected coarse granularity: a single bit in the vector represents the number of locks that can be stored in a single protocol processor cache line. This allows the timeout bit vector storage to be very compact, and yet the timeout checking for the group of locks is efficient since they are resident in the cache at the same time.

In the current FLASH lock implementation, a 128 B protocol processor data cache line can hold sixteen locks, thus a single bit in the vector indicates that one or more of the sixteen locks on that line received a token grant recently. By extrapolation, a bit vector consisting of only a single PP cache line (128 B = 1024 bits) can represent 16 K FLASH locks. Figure 6.16 illustrates this bit vector approach. For the rest of this section, we describe an implementation for 16K locks; larger implementations can be generated simply by scaling the bit vector.

The timeout mechanism is invoked periodically by the Idle Handler in MAGIC, described in Section 2.3.5. We anticipate providing generous timeouts, on the order of the process time slice of several milliseconds, which keeps timeout checking overhead very low. Once invoked, the timeout

```
InitTimeoutIteration()
{
   if (!timeoutCheckDone) {     /* Verify prior round finished */
      return;                   /* Start new check next time */
   }
   copy AgingTimeouts to CheckingTimeouts;
   copy InstallTimeouts to AgingTimeouts;
   clear InstallTimeouts;

   Initialize timeout checking;
   timeoutCheckDone = 0;
   SWQSchedule(TimeoutChecker);
}

TimeoutChecker()
{
   scan CheckingTimeouts for nonzero bits by doublewords;
   if (entire mask is zero) {
      timeoutCheckDone = 1;
      SWQUnschedule();
      return;
   }

   check = first nonzero doubleword in CheckingTimeouts;
   pos = FindFirstSetBit(check);

   /* Check one line of locks at most, to share PP */
   foreach lock on line given by (check,pos)
      if (lock.Token && !lock.Seen) {
         /* Found one. Clean up as appropriate:  */
         if (lock.Requested) {
            Verify network queue space;
            lock.Token = 0;     /* Steal lock away */
            Send lock to external requester;
         } else {
            lock.Seen = 1;      /* Allow external requests */
         }
      }
   }
   ClearBit(check,pos);
   SWQReschedule();
}
```

Figure 6.17: Pseudo code for the FLASH lock timeout mechanism.

mechanism schedules itself to execute from the software queue repeatedly until its processing completes (the duration varies based on the intensity and locality of lock activity). The pseudo code for the timeout mechanism appears in Figure 6.17. It refers to three bit masks of the nature described above:

`InstallTimeouts` This vector is where new token grants are recorded. The grant handler is responsible for setting the bit corresponding to the lock being granted.

`AgingTimeouts` when a new timeout round begins, `InstallTimeouts` is copied here. This assures that a lock has at least one full timeout period after its grant.

`CheckingTimeouts` This vector is used to perform the actual timeout checking.

The space overhead of the mechanism is constant and extremely low: only three cache lines to store the three bit vectors. The advantage of the vector approach is the varying granularity at which locks can be scanned. The first scanning occurs on doublewords, each of which represents a full 1 K locks (64 bits*16 locks/bit). This allows inactive lock ranges to be eliminated very quickly. Once an active doubleword is found, the Find First Set Bit (FFSB) instruction can be used to scan for activity in hardware at a 16-lock granularity. Set bits cause examination of a group of 16 locks individually. Locks found to have received a token grant without a subsequent request lose their Seen-bit protection or are sent to the next requester, if present.

Retrying Token Requests

Multiprogramming has still another unfortunate effect on locking. Requests for a lock token are tracked based on the *node* on which they originated. Just as in shared memory locks, there is nothing to prevent one process from requesting the lock, and then before the token arrives multiprogramming causes another process to execute. If the second process desires the same lock, it may consume the token that arrives for the other process's request. In fact, if the lock is requested elsewhere, the token may also be lost before the first process ever has a chance to acquire the lock.

The risk is that the first process will not realize its token was consumed by another process and will blithely spin waiting for it. We solve this problem by clearing the Requested bit when the token is lost. As part of its spin loop, the `FLASHLock` routine must check the Requested bit to see if its request is still active. If it sees it clear, it re-requests the token. This mechanism is also important to support the timeout mechanism: If a process is preempted just after its lock request and its token arrives then times out, this technique assures it issues a new token request if needed when it later resumes execution.

This issue elucidates the exact granularity of the fairness guarantee FLASH locks provide: the *processor* is granted the lock in a fair manner. FLASH locks, like shared memory locks, have no way of controlling which process on that processor actually acquires the lock after the token arrives at a processor. Moreover, any solution attempted at a low level runs the risk of causing higher-level problems such as priority inversion, so this issue is better solved, if needed, by a higher-level mechanism [SRL90].

6.1.7 Protocol Handlers

One of the clear successes in the lock protocol is its very compact protocol code. This allows the lock protocol to share the PP instruction cache with cache coherence much more effectively than the FLASH memory copy protocol, for example. Table 6.1 enumerates the handler components of the FLASH lock protocol, which all told amount to less than 3 KB. This compact design was enabled by two key design decisions.

Table 6.1: Summary of the FLASH lock protocol handlers.

Handler Name/ Handler Size (bytes)	Description
LockStatusRead 104 B	The "cache miss" handler for lock protocol state requests from the processor. It does *not* cause any token movement. This handler provides exclusive ownership even if a read-only copy is requested, eliminating the need for upgrades and replacement hints in the protocol.
LockWriteback 208 B	The handler for lock protocol state writebacks from the processor. Should a lock line be replaced from the cache, it collects the current values of the locked and seen bits and updates the MAGIC lock state.
PIGetLockToken 200 B	Requests a token needed by the processor but found to be absent. This is invoked explicitly by a processor PPR.
PIUnlock 152 B	Invoked explicitly by a processor PPR, this handler accepts a token from the processor following an unlock that noticed an external request was pending. This handler sends the token to the next processor in line.
NIRequestLockToken 1272 B	This handler serves two roles: it accepts new requests for the lock token (if invoked at the lock home), and requests to steal a lock token away (if invoked elsewhere). These two functions form the majority of the lock protocol's token movement functionality, thus this handler is the largest.
NIGrantLockToken 128 B	Accepts an incoming token from the network, invalidating the related lock state, if cached, to cue the processor.
SWRequestLockToken 528 B	Executing from the software queue, this regenerates a queued lock request which could not be handled due to outgoing network queue limitations.
Total size: 2880 B	(includes a support subroutine not listed)

First, by referring to locks in an alternate address space, a custom coherence protocol could be used that was able to avoid the complexities of integrating with the full-fledged coherence protocol. As Table 6.1 shows, just two handlers (LockStatusRead and LockWriteback) are needed to provide the majority of coherence functions.

Second, token movement was largely separated from coherence functions, which reduced the total number of cases to be handled. Instead, five handlers have the primary responsibility of moving tokens (there is a minor interaction with coherence since they are required to notify the cache upon token arrival).

6.1.8 Lock Performance in Isolation

In this section we evaluate the performance of the lock protocol using a range of metrics. We begin as we did in Chapter 5 by considering the performance at a very low level, examining the characteristics of FLASH locks and other alternative locking primitives. We perform this study of locks in the context of microbenchmarks that compare this range of primitives in a controlled manner. This allows us to isolate the performance differences between the primitives in various operating situations. Later, in Section 6.3, we consider the performance impact of the locks in the context of applications.

We do not attempt to compare against each of the wide range of alternatives for implementing locks in software. Rather, we focus our comparison of FLASH locks against several key choices to allow us to isolate and explain the major effects. Other researchers have considered the breadth of software alternatives in detail in previous publications [MCS91a, MLH94, KBG97]. Specifically, in our analysis of locking performance we consider the two different lock primitives we described earlier in this chapter: LL/SC-based locks and MCS locks. We include the LL/SC lock in this study despite it being a fairly simplistic lock implementation because it is nonetheless widely used in applications, and because load linked and store conditional operations are commonly provided by many processor manufacturers. We study MCS locks since they provide essentially the highest performance available from processor software techniques alone.

The simulation environment used in this chapter is the same one described in Sections 2.4.2 and 5.4.1. The only difference is that this study was performed after the switch to the CrayLink network [Gal96], so we model it in our simulations.

Non-Contended Lock Acquisition

We first study non-contended locks, ones which are available when requested, using the acquire latency metric introduced in Section 6.1.1. We measure non-contended acquire latency using a hand-crafted microbenchmark that isolates the test to eliminate other activity in the system. The results are shown in Table 6.2. As we described, the acquire latency varies based on whether the nodes involved are the home node where the lock is allocated, so we explore the permutations of whether the previous holder and new requester are the home node of the lock. The remote to remote case considers the transfer between two different nodes, neither of which is the lock home. Finally, since these locking primitives permit caching, we consider the acquire latency when the same node re-acquires a cached lock it previously held. Cached re-acquire times are independent of where the lock is allocated.

As expected, the results show that shared memory techniques allow low acquire latency for locks which are available. In fact, the performance of LL/SC locks and MCS locks is nearly identical in

Table 6.2: Acquire latency for an available lock (microseconds). The results are shown for the three different locking primitives and with different combinations of lock requesters. The FLASH Aggressive variant is explained in the text.

Previous Holder	Lock Requester	Locking Primitive			
		LL/SC	MCS	FLASH Normal	FLASH Aggressive
Lock Home	Remote	1.13	1.17	2.64	2.25
Remote	Lock Home	1.06	1.11	2.43	2.06
Remote	Other Remote	1.45	1.50	3.01	2.64
Same node re-acquire		0.03	0.07	0.12	0.50

these cases. For available locks, MCS locks perform little additional work as compared to LL/SC locks, which is one of the benefits of the MCS technique.

Unfortunately, FLASH locks (Normal) are much slower than shared memory locks in the acquire time metric for two specific reasons. First, the initial request for a FLASH lock takes longer than that for shared memory locks. To acquire a FLASH lock, the processor first requests the lock status and if it finds the token missing, only then does it request it via a PPR. This introduces an extra round trip as compared to shared memory locks which immediately requests the lock from the initial miss.

As we explained earlier, the presence of speculation prevents us from launching the token request from the lock status read handler. Were the token request able to be launched there it would decrease FLASH lock acquire latency by about 0.45 μs. One alternative exists in software that works correctly despite processor speculation: within the lock library call we could issue the token request blindly *before* reading the lock status, essentially prefetching the token. The performance of this approach is listed under FLASH "Aggressive" in Table 6.2. When the token is not present, this succeeds in reducing the acquire latency by approximately 0.4 μs. In the case of lock re-acquisition, the token request handler correctly ignores the request if the token is already present, however it still ties up the protocol processor. This delays the subsequent lock status read if the lock is not cached, increasing re-acquire time by about the same 0.4 μs, or delays other reads such as accesses within the critical section. The choice between these two flavors depends on the expectation of whether the lock may be reacquired; these results suggest extending the FLASH lock API to provide this version for cases where re-acquisition is unlikely.

Another approach to handling speculative requests from the processor is merely to ignore the problem and request the token when the processor issues the lock state read. The timeout mechanism in the protocol ensures correctness should speculation occur, though the latency could be as high as several milliseconds. If speculative requests are sufficiently infrequent, this approach might increase overall performance.

The second reason FLASH locks are slower is caused by the token movement protocol's handling of lock state cached by the processor. When a node's request for the token reaches the current token holder, several things must happen. First, the token holder must examine the MAGIC state to verify that it has the token and expects to able to supply it. There are a number of cases in the protocol, largely due to queueing, so this state check is nontrivial. Then, if it finds the lock state is cached by the processor (typically the case) it must extract the state from the cache to see if the lock is currently asserted. This request must wait for the processor cache's reply and then examine the result, unlike the cache coherence protocol's forwarding which can launch the message early and allow MAGIC to stream the data out the network as it arrives from the processor. These effects combine to make the token request handler (NIRequestLockToken) inefficient for available locks, about 1.0 μs overall. Optimizations to this handler may improve the performance of the state check phase slightly, but the processor cache request is fundamental and accounts for a significant fraction of the handler.

Contended Lock Acquisition

Despite the slow performance for uncontended locks, FLASH locks perform extremely well under contention, i.e. when locks are found to be held by another processor when requested. The two effects in particular which affect acquire latency do not impact contended lock hand-off latency. First, the extra latency of token requests is hidden because the lock is held and cannot be supplied immediately. Second, the token request is enqueued at the current holder, thus the cache access is not in the critical path. Instead, when the lock is later released it is handed off directly to the requester at very low latency.

To evaluate contended lock performance, we use a microbenchmark that reflects a number of processors making one update each to a lock-protected data structure. This benchmark, like the previous one, carefully eliminates other activity in the system to isolate synchronization performance. In fact, to reduce even the interference from accesses in the critical section, and to intensify the contention to the theoretical worst case, we eliminate the data structure modification itself. Instead, each processor merely acquires the lock and releases it immediately. Later in this section we consider a different microbenchmark that simulates a lock being repeatedly acquired and released, to study the performance of contended accesses arriving in a stochastic fashion.

In this benchmark, one processor (the lock home) acquires and holds the lock for a long time while all the other processors request the lock and find it busy. Then the home finally releases the lock and the benchmark begins, continuing until all processors are able to acquire and release the lock a single time. In each of the lock primitives, steady state is reached during the initial wait phase for the lock, which has two different forms: In the case of LL/SC locks, each node caches the lock value indicating it is already held. When the lock is later released, these nodes all rush to the home.

Table 6.3: Two results from the contended locking microbenchmark: hand-off latency and all-acquire latency (microseconds).

Total Processors	Hand-Off Latency (single lock)			All-Acquire Latency (NumProcs − 1 locks)		
	LL/SC	MCS	FLASH	LL/SC	MCS	FLASH
2	1.7/1.1	0.8	1.0	2.8	1.5	1.4
4	3.3	1.5	1.0	10.5	9.4	4.5
8	6.3	1.6	1.0	28.5	19.4	9.5
16	10.1	1.6	1.0	77.4	40.8	18.5
32	17.0	1.6	1.0	189.8	85.9	39.3
64	30.4	1.6	1.0	449.2	174.9	75.7
128	57.4	1.6	1.0	1214.4	398.6	159.4

The queued lock primitives, on the other hand, construct a queue of waiters during this time when the lock is unavailable, and then smoothly transition the lock down the list.

From this benchmark we can measure two useful quantities. The first is *hand-off latency*, introduced earlier, which measures the time from the initial release until the first waiter succeeds in acquiring the lock. This metric is useful to isolate the overhead from contended hand-off with no other interference. The second metric is the total duration of the benchmark which we refer to as *all-acquire latency*. Note that for a given number of processors, n, all-acquire latency measures $n - 1$ total lock acquisitions and releases, since the home node holds the lock to allow the initial condition to stabilize. While hand-off latency is a useful metric for isolating one effect, all-acquire latency also includes other costs such as unlock overhead, and thus reflects the realistic performance of a series of acquisitions in a row.

Table 6.3 shows the results of the benchmark for the primitives we study, over a range of machine sizes. We see that both MCS and FLASH achieve essentially constant hand-off latency, since there is no contention within the release process as a result of the pre-constructed queue. LL/SC on the other hand degrades in performance as more nodes rush for the lock when it is released. LL/SC locks, unlike the other techniques, perform better when next lock holder is the home. The 1.1 μs hand-off latency corresponds to this case.

All-acquire latency results are also presented in Table 6.3; in addition, Figure 6.18 plots all-acquire latency divided by the number of lock acquisitions ($n - 1$) the benchmark executes. In the figure, a flat horizontal curve corresponds to ideal scalability—a lock with performance independent of machine size. We see that both MCS and FLASH locks essentially achieve this ideal. The y-value of the curve shows absolute per-lock time, which for FLASH (about 1.3 μs) is lower than that of MCS (2.8 μs), as expected from the hand-off latency results. Note that the difference between the two is larger than difference in the hand-off latency results since the all-acquire time includes unlock

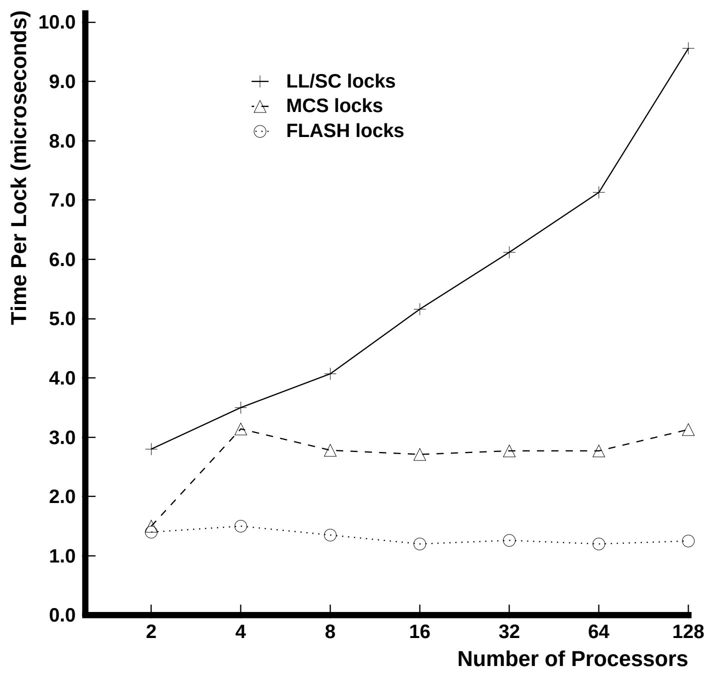


Figure 6.18: Results from the high contention benchmark under simulation. The plot illustrates average time in microseconds per lock, as a function of machine size (*all-acquire latency* divided by NumProcs − 1).

operations, which are also slower for MCS. LL/SC locks are inferior in every respect: the absolute overhead is higher, and the curve's positive slope reflects increasing overhead as the machine scales.

One other effect the hand-off latency and the figure both show is a jump in latency for MCS locks from 2 to 4 processors and then essentially constant performance from then on. This arises from the MCS unlock operation which is able to do less work when it reaches the end of the queue. For two processors, the queue is only one-long, so this effect is visible there.

Contended Lock Acquisition (Real Machine)

Since the results for contended lock acquisition were generated from simulation, it is interesting to verify our scalability conclusions using a real system where possible. Though a real FLASH system is not yet available, we are able to evaluate the scalability of the LL/SC and MCS lock primitives using a 16-processor Silicon Graphics O2000. The O2000 provides a reasonable comparison point since it shares with FLASH the same processor and network. The major difference is that the O2000 uses a proprietary hardware node controller in place of the MAGIC chip, and our system only has 16 processors. Unlike simulation which allowed us full visibility into the processor, measuring individual lock times accurately is not possible on the real system. However, we are able to measure

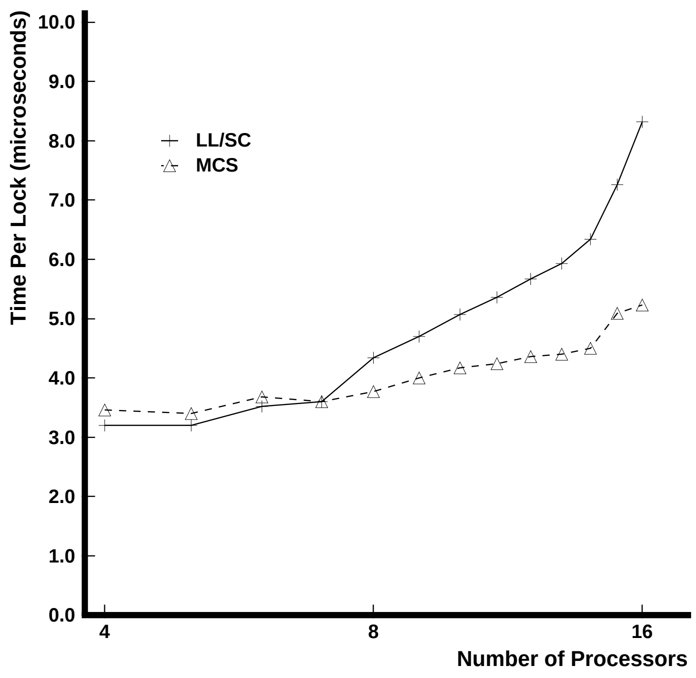


Figure 6.19: Results from the high contention benchmark running on the Silicon Graphics Origin 2000 multiprocessor. The plot illustrates average time per lock acquisition (*all-acquire latency* divided by NumProcs $- 1$)

the all-acquire latency with reasonable precision using the 800 ns granularity hardware counter in the O2000.

The O2000 results from the same contended lock benchmark are shown in Figure 6.19. We see that the times per lock are uniformly slower than those from simulation (by a factor between approximately 1.5–2). The absolute overheads for these primitives is a function of many different effects, including the detailed processor and cache controller timing and so it is not surprising that our simulator does not exactly match.

As for scalability, we see similar trends to those encountered in simulation. MCS time is nearly constant, but does have a clear increase as more processors are involved. Despite our careful attempts to isolate this benchmark while executing on a real system, an unknown effect or source of interference is clearly at work here, unlike simulation where we are able to achieve complete isolation and perfect visibility. LL/SC lock time increases as before, though in this case we see significant performance degradation even at 16 processors, amplifying all the more the need for queued locking even for small machines.

Stochastic Contended Lock Acquisition

By design, the contention microbenchmark described in the previous section is very controlled, to carefully isolate the performance of lock hand-off under contention. It is somewhat artificial, however, since it allows the entire queue of waiters to be constructed in advance.

In this section we study a less contrived high-contention microbenchmark that interleaves lock requests throughout its execution, similar to the one used in [KBG97] and [LA94]. This benchmark is less convenient for isolating the exact performance costs because of the random nature of the events it contains, but is somewhat more representative of the real performance an application might encounter.

> [The] microbenchmark . . . accesses a critical section in a loop repeatedly (the benchmark accesses the critical section a total of 3,200 times; these accesses are distributed evenly among the processors). Once in the critical section, a processor waits 800 cycles before releasing the lock (this stall simulates access to, and computation of, protected data). After release, the releasing processor waits for a random time selected from a uniform distribution. The mean of the distribution is five times the critical section delay (4,000 cycles) [KBG97].

We attempt to tune the benchmark to match the delays used by these prior researchers, though our different time and processor modeling is likely to cause the absolute magnitude of the results to differ. In particular, we retain the factor of five difference between critical section length and mean acquisition interval, in an attempt to provide the same degree of contention as previous studies.

Figure 6.20 illustrates the results for this benchmark, expressed as its total execution time. As before, LL/SC scales poorly, causing lock throughput to degrade quickly. For MCS and FLASH locks the execution time is essentially constant over a wide range of machine sizes (8–128 processors), which corresponds to desirable lock throughput independent of machine size. As the machine scales, execution time does increase slightly as a result of requests to rejoin the lock queue. Overall, the performance of MCS and FLASH do not significantly differ. Since a large fraction of the benchmark execution time is spent idling inside the simulated critical section, only a small fraction of the time is actually spent in transitioning locks, and thus speedup is limited by Amdahl's law.

These microbenchmarks foreshadow the results from application simulations we present in Section 6.3. In some cases we find that extremely high contention for locks limits performance, and in those situations FLASH locks show gains from improving communication. In cases where the contention is less severe, or represents only a small fraction of the application, we find that improvements from FLASH locks are milder and tend to be similar to MCS, which also achieves the bulk of the gains as compared to LL/SC.

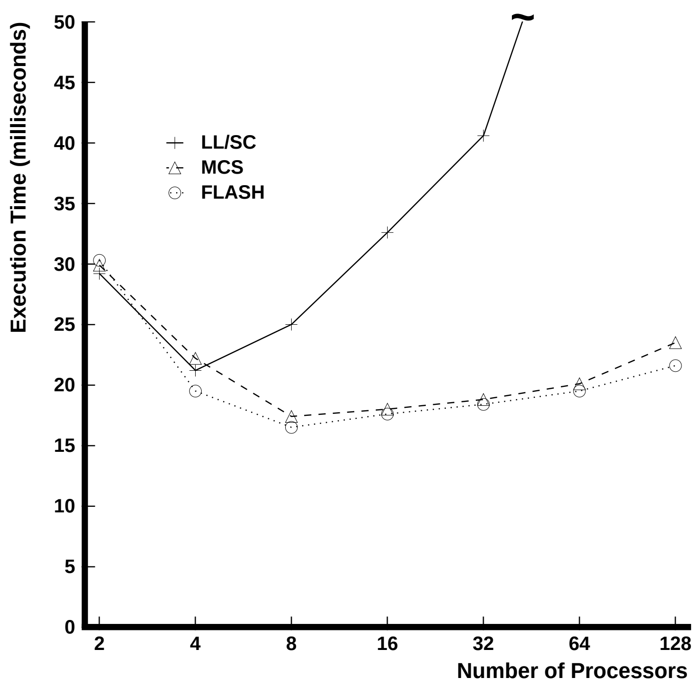


Figure 6.20: Performance of the stochastic contended lock microbenchmark under simulation. The plot illustrates overall execution time of the benchmark, in milliseconds.

6.1.9 Related Work

Locking synchronization has been studied in many different contexts and by different approaches. We present the related work most relevant to FLASH locks, and to the large scale parallel processing environment we study. We focus in particular on one primitive, QOLB, as it is shares many common characteristics with the FLASH lock. We also consider approaches to attack the artifactual communication problem by avoiding locking altogether.

Graunke and Thakkar [GT90] present a performance comparison of synchronization primitives for a bus-based multiprocessor. Their analysis advocates the use of back-off for locks experiencing mild contention, or queue locks when the contention is significant. In their experiments using a simple benchmark, queue locks were superior in most metrics past 5–10 processors on a Sequent Symmetry.

In two related papers, Anderson [And89, And90] studies the performance of spin lock primitives. These studies focus primarily on different kinds of delays between lock retries, designed to reducing contention heuristically. The papers find that the exponential back-off technique, in which a processor's maximum wait time increases exponentially as contention repeats, is a practical approach for its simplicity and performance. Anderson also presents a queue lock primitive, but it is inferior to the MCS lock in many respects.

Earlier in this chapter we described in detail the MCS lock primitive introduced by Mellor-Crummey and Scott [MCS91a, MCS91b]. Since their original papers, a number of researchers have studied their primitive and described modifications to improve its performance in specific situations. Herlihy et al. [HLS95] present a range of techniques for software counting including a custom version of MCS locks for that purpose.

Magnusson et al. study the MCS lock primitive closely using analytical techniques and propose two new lock primitives, the LH lock and the M lock [MLH94]. Their analysis demonstrates that under contention the MCS lock release requires an additional memory read as compared to LH, but that LH requires additional acquire traffic when locks are not reacquired by the same processor successively.

The M lock uses an ingenious approach to optimize performance still further: packed with the word storing the lock's status it indicates the last writer of the lock. This allows the M lock release routine to avoid a global write if a lock queue did not form when the lock was held, while still allowing the lock to be released in a single operation if one did. The M lock thus improves global traffic in some regimes as compared to MCS, which is especially valuable as systems scale, at a cost of increased lock primitive code length, which may detract from its advantages in cases where critical sections are small or contention is low. Kägi et al. [KBG97] suggest that the extra cache miss can be eliminated from the MCS lock by collocating the lock indication with the next pointer. We use their approach in our implementation of MCS locks.

An approach called *reactive synchronization* is proposed by Lim and Agarwal [LA94] to address the tradeoff between lock regimes. Instead of relying on a single lock variant for all situations, they describe how the lock primitive can detect contention (from failed acquire attempts) and change to a more resilient protocol. This allows the latency advantages of light protocols in low-contention cases with the throughput benefits of protocols like MCS as contention increases. Though we do not study this approach in FLASH, the ability to select different protocol characteristics within MAGIC makes this approach well-suited to our environment. In particular, this technique may be useful to address the performance limitations of non-contended FLASH locks, selecting a different primitive in that regime instead. We reflect on the use of reactive synchronization-like techniques in FLASH in Section 6.4.

The DASH system [Len92] provided an special facility in the protocol to implement a kind of queued lock by extending the coherence protocol. Lock and unlock operations in DASH are accessed through uncacheable alternate memory spaces but lock values otherwise appear as normal memory locations, both similar to our approach. DASH provides an operation called a *granting unlock* which invalidates only a *single* random waiter for the lock instead of invalidating all processors as usual.[3] This avoids a rush of requesters following an unlock. DASH also provides an operation to force a

[3]A random waiter is released because the bit vector coherence directory structure of DASH does not maintain a notion of ordering that could be used to provide FIFO access to the lock.

line to its home, which is used at a lock release so the next holder's request may be satisfied at home. This combination of support helps to reduce lock contention, though it lacks the ability to transfer locks directly from holder to holder provided by FLASH locks and QOLB.

Falsafi et al. [FLR+94] advocate the use of application-specific communication protocols to increase performance using knowledge of an application's characteristics. They describe a range of custom protocol modifications including changes to the coherence model and granularity and optimized synchronization. They refer to a message-based lock protocol built on top of the Tempest interface that optimizes lock hand-off as compared to MCS locks. Like FLASH locks, their performance gains come from matching synchronization traffic to the underlying primitive rather than layering on top of shared memory. Their paper does not describe the protocol itself, but their results agree with our findings that custom synchronization support in this environment can show significant gains.

One important observation is that their paper describes a fairly broad range of custom protocol optimizations, among them coherence protocols that include update functionality. Several of these optimizations are not possible in the current FLASH prototype due to its processor interface and system design choices we described previously. For example, update operations are not permitted by the R10000 bus interface, and MAGIC does not implement a remote access cache to hold updates in the memory system (though a RAC could be implemented in software).

Their study focuses on the Tempest interface running on the Blizzard-E system (described in [SFL+94]), which allows them a wider range of operations but at far lower performance than a dedicated hardware implementation. A hardware Tempest implementation, such as Typhoon [RLW94] would likely entail implementation-specific restrictions similar to FLASH.

Distributed Queueing: SCIand QOLB

Previous research work has proposed the use of distributed queueing for both cache coherence protocols as well as locking. One use of distributed queueing is to store the sharers of a cache line in the Scalable Coherent Interface (SCI) coherence protocol [Mic93]. This approach differs from the centralized sharer list approach of FLASH, Alewife, and many others. Despite our positive results using distributed queueing for locks, Heinrich found that an implementation of the SCIcoherence protocol for FLASH was very complex [Hei97]. In particular this complexity arises because SCI-maintains distributed *doubly-linked* lists, and must be able to remove sharers from anywhere in the list to handle cache line replacements.

The QOLB synchronization primitive (*Queue On Lock Bit*, originally called QOSB) was introduced by the Wisconsin Multicube project, and then subsequently used to support locking in the SCIprotocol [GVW89, KBG97, Goo97]. The most detailed explanation of QOLB, including details

```
acquire(line)
{
   do {
      while (QOLB(line))
          ;                      /* Spin */
   } while (Test&Set(line.lock));
}

release(line)
{
   Unset(line.lock);
   UnQOLB(line);
}
```

Figure 6.21: Pseudo code for a lock implemented with the QOLB primitive, taken from [AGGW92].

of one proposed implementation within SCI, is provided by Aboulenein et al. [AGGW92]. Conceptually, the QOLB approach is very similar to FLASH locks. In fact, the basic QOLB interface can be implemented directly on the existing FLASH lock primitive; we describe the features of QOLB below to explain the correspondence.

Figure 6.21 shows how locks are implemented using the QOLB primitive. In short, QOLB performs three functions: *(i)* It indicates the processor's intent to acquire the lock, and requests the lock be fetched to the node. The parallels the token request PPR issued within the FLASH lock library. *(ii)* It returns a result immediately to indicate if the line is currently present on the node. The value it returns corresponds to the bit in the FLASH lock status line that indicates token presence. The while loop in Figure 6.21 corresponds to a loop inside `FLASHLock` that spins waiting for the token to arrive. *(iii)* It enqueues the requester on an SCIwaiter queue (using a special QOLB mode) to provide FIFO access to the line and permit direct holder-to-holder hand-off. FLASH locks provide both these characteristics by default for locks, since locks use a dedicated protocol instead of building on conventional shared memory.

Since QOLB is non-blocking, it can be used to prefetch the lock before it is actually needed, an optimization studied in detail by Woest and Goodman [WG91]. The FLASH lock API does not currently provide this mode, but it could be supported merely by separating out the token request from polling or spinning for lock arrival. The downside to prefetching in both QOLB and FLASH locks is that once the lock (or in our case, lock token) is granted to a node, other nodes are prevented from acquiring the lock (during a timeout window) even if it has not actually been asserted. This suggests that lock prefetching must be used carefully to avoid lengthening the effective critical section duration.

The QOLB primitive does *not* acquire the lock itself, like the token request in FLASH. Both protocols acquire the lock once the token is present (or QOLB returns success) using an atomic memory update primitive such as Test-And-Set or LL/SC. Since QOLB locks operate on normal memory locations, use of the QOLB primitive itself is *optional*, since the atomic operation still operates correctly (though at lower performance) using normal shared memory mechanisms if the line was not previously fetched. In FLASH locks, on the other hand, the token fetch is *required* since they do not fall back to the normal cache coherence protocol like QOLB.

This design difference leads to two other differences between the primitives. First, when a lock is fetched in QOLB, it carries with it a line of data, which appears in the cache with the lock. This prefetching-like *collocation* effect can be very beneficial, particularly if the lock protects a small data structure. FLASH locks cannot easily provide this feature because locks are treated separately from cache coherence. Though this is a minor drawback to our approach, keeping the lock protocol separate from the coherence protocol is one of the reasons that it is so compact, and thus performs so well.

On the other hand this difference causes QOLB to suffer from an unfortunate flaw that occurs if a lock line is replaced or if a processor tries to acquire a lock not already present in its cache.[4] In those cases SCItakes over and performs a *normal* exclusive read, which destroys the queue of waiters that accumulated. This does not violate mutual exclusion, but it does violate the FIFO ordering guarantee and requires the queue be reconstructed (very likely in a different order). FLASH locks prevent this failure mode since atomic operations only act on the local copy of the lock line, with the queue being maintained reliably by the lock protocol on MAGIC. The timeout solution in FLASH locks is also superior to that of QOLB because their solution steals the lock in a manner that often results in this queue breakdown situation. If a FLASH token holder takes too long and the token is stolen, only that node is forced to request the token again.

QOLB has been evaluated using an analytical approach both for the basic primitive [AGGW92] and focusing on the prefetching effect [WG91]. More recently, QOLB has been studied under simulation to compare it against other locking alternatives [KBG97]. At the level of detail of these studies, QOLB and FLASH locks perform similarly, except that we do not explore lock prefetching and can not readily support the ability to collocate data with locks given our current state implementation. The results of these studies show similar trends to ours, though the simulated microbenchmark results predict larger gains from QOLB over MCS than we experienced with FLASH locks. This difference may arise from remote memory costs being higher under SCI(reducing the apparent performance of MCS locks) or from differences in our microbenchmark timing.

In a recent technical report, Kägi describes a purely software implementation of QOLB called SOFTQOLB [KG98]. This implementation uses the Blizzard run-time system [SFL$^+$94] to provide the Tempest interface on top of a cluster of commodity workstations. Since this environment differs considerably from the one we study, the results of that paper are not readily comparable to ours.

[4]This can occur due to migration, multiprogramming, and from a race condition between `QOLB` returning success and the atomic update primitive.

Avoiding Mutual Exclusion

One alternative to improving locking performance is to avoid the requirement for mutual exclusion altogether. Researchers have explored several different approaches to this problem; in general they require specialized hardware support or application software modifications. We discuss the first of these primitives, Fetch-and-Op, in more detail in Chapter 7; a more detailed study of them is beyond the scope of this dissertation.

The NYU Ultracomputer and the IBM RP3 (Research Parallel Processor Prototype) projects introduce a synchronization operation called Fetch-And-Op [GLR83, GGK$^+$83, PBG$^+$85, FG91]. Fetch-and-Op performs atomic updates at the memory location itself using specialized hardware, thus avoiding the pinging of contended lines between different processor caches. These projects also explored a specialized interconnect called a *combining network*. The combining network switch, described in Section 7.2.1, improves performance further in high contention scenarios by merging different processors' requests as they traverse the network, reducing multiple memory updates to a single one. Fetch-and-Op is ideally suited to a class of simple critical sections such as dynamic work distribution, it can also be used in concert with some of the techniques described below to support more complex updates. The Cray T3D implemented a single Fetch-and-Increment register per node to support work distribution; the T3E extended this support to operate on memory and added Fetch-and-Add functionality as well [Cra93, Sco96].

Wait-free (also called *Lock-free*) synchronization techniques allow cooperating processes to assure consistent data structure updates without using a lock-protected critical section. This can be achieved by a number of software or hardware techniques. Herlihy [Her90] describes software techniques that can be used to avoid mutual exclusion, including one technique in which processes "help" each other so that updates by one processor do not cause another processor's pending operation to become inconsistent.

More recently, Herlihy et al. [HM93a] describe hardware support to enable lock-free characteristics at much lower overhead. This support, called *transactional memory*, provides the ability to atomically update several memory locations at once and to detect other updates that render previous accesses inconsistent, similar in concept to the semantics provided by transactions in a modern relational database. This approach is particularly appropriate for snoopy bus-based machines or cache-coherent machines with sequentially consistent memory. In an environment like FLASH containing distributed memory and weak memory consistency, transactional memory requires additional memory fence operations to provide the appearance of sequential consistency.

As an alternative to using hardware support, Shavit et al. [ST95] present STM, a software-based implementation of transactional memory using LL/SC. Their implementation constructs a table describing pending memory modifications, somewhat similar to the information stored in hardware in the original paper. Their experiments using STM on the MIT Alewife machine show that it outperforms other non-blocking software synchronization approaches, but that STM and other non-blocking techniques are inferior to queue locking methods such as MCS Locks.

6.2 FLASH Barriers

The second synchronization primitive we study is barriers. The semantics of a barrier is that a processor which arrives at a barrier may not proceed past it until all other processors reach the barrier as well. This functionality is often nicknamed a *rendezvous* (meeting), since it forces the execution of processors to meet before proceeding further.

Barriers are typically placed in applications with phase-based characteristics. There the barrier is used to assure that no processor proceeds to a subsequent phase until all have finished the current one. This is particularly important if values computed by processors in phase i will be read by other processors in phase $i + 1$. Without the barrier, processors might read partially completed results from phase i if, for example, a load imbalance has made the phase durations on each processor differ.

High performance barriers are important in some classes of applications that synchronize frequently. Scott [Sco96] describes a proprietary meteorological application in production use on the Cray T3E in which a 128 processor system requires a barrier every 200 μs. The T3E can achieve a barrier in 15 μs in software using special message passing features, or in approximately 2 μs using special barrier hardware support. We show later that even a highly optimized barrier implementation on top of cache-coherent shared memory can take 30 μs for 128 processors. This section presents a barrier protocol for FLASH that can achieve the same synchronization in 4.7 μs, requiring no additional hardware beyond the facilities already present in FLASH for cache coherence.

The outline of this section is similar to the section on locks. We begin in Section 6.2.1 by describing a methodology for analyzing barrier performance. Then, in Section 6.2.2 we present several conventional barrier implementations, to illustrate the strengths and weaknesses of these approaches. We present the application programming interface of the new FLASH barrier in Section 6.2.3, and then describe its design and implementation in Section 6.2.4. We show the code organization and size in Section 6.2.5. We study the performance of FLASH barriers in isolation in Section 6.2.6 (application results for locks and barriers are presented later in Section 6.3). Finally, Section 6.2.7 summarizes some additional related work in this area.

6.2.1 Metrics for Evaluating Barrier Performance

We begin our analysis as we did for locks by describing the desirable characteristics for barriers. Then, since barriers are a more complicated synchronization primitive than locks we establish a framework that can be used to compare different barrier implementations consistently.

The critical performance characteristic of barriers is achieving low overhead and high throughput (i.e., barriers per second). Barrier overhead appears as the dead time during which some or all the processors are forced to wait at the barrier even though all have arrived and satisfied the required rendezvous. Conventional barrier implementations fail to achieve low overhead for many of the same reasons as locks, in particular artifactual communication within the barrier primitive. Moreover, since barriers necessarily involve all the processors, contention based performance degradation can readily occur.

Low barrier overhead has scalability benefits for parallel applications. Since synchronization represents time not spent doing useful work, applications are typically crafted to increase grain size and decrease synchronization frequency. Reducing barrier overhead may enable finer grain synchronization that eases application design, enables scalability in new application classes, or increases the machine size to which existing applications can productively scale.

Besides hand-written applications, finer granularity synchronization can also benefit automatically parallelized applications. The SUIF compiler [HAA$^+$96] parallelizes applications by identifying independent threads (e.g., DOALL loops in FORTRAN), executing them in parallel, and then synchronizing the processors with a barrier. The parallelism detected by the compiler is often very fine, and thus reducing synchronization overhead is important to increase the fraction of loops which can be sped up by parallelization. Due in part to high synchronization costs, coarse loops are currently the primary focus of these compilers [BAR96, BAM$^+$96]. Section 6.2.4 describes how FLASH can support a unique kind of barrier tailored specifically to the structure of compiler-parallelized applications.

Barrier Nomenclature

Isolating the performance of a barrier is more difficult than in locks for two primary reasons: *(i)* barriers fundamentally require the participation of multiple processors, and so there is more parallelism involved in the operation, and *(ii)* there are some interactions between barriers and the application that executes them. To isolate these effects, we define two metrics below that serve to analyze the two main components of barrier performance while at the same time reducing the interference from application behavior.

We begin by establishing some nomenclature for barrier operation; Figure 6.22 illustrates a generic barrier instance. For this analysis, the actual processor numbers are not significant, we focus instead on the relative indices with which processors join and leave the barrier.

On the left side of the figure are the arriving processors, named symbolically by the order in which they *join* the barrier: $J_1 \dots J_n$. Our analysis considers a processor to have joined the barrier when it *begins* execution of the barrier primitive in the program. Furthermore, we say that arrival J_i occurs at time t_{J_i}, thus the array of arrival times is denoted $t_{J_1} \dots t_{J_n}$.

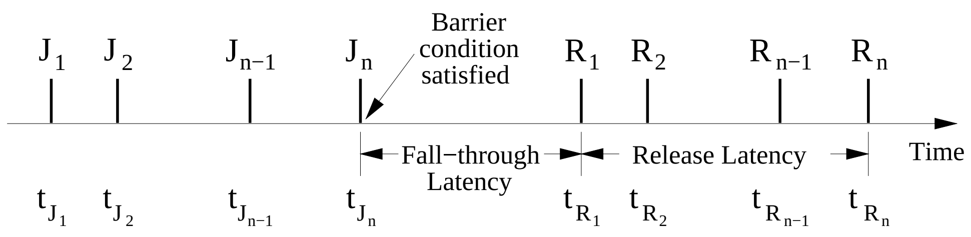


Figure 6.22: Barrier nomenclature illustration, showing fall-through latency and release latency.

Similarly, the right side of the figure names the processors being *released* from the barrier: $R_1 \ldots R_n$. Processors are released when they *finish* executing the barrier primitive and resume application processing, designated in a similar way: $t_{R_1} \ldots t_{R_n}$. Note that the join and release orders need not be correlated, i.e., the first joining processor J_1 need not be the one released first, R_1, etc.

Barrier Fall-through Latency

The first metric of barrier quality, as illustrated in Figure 6.22, is referred to as *fall-through latency*. Fall-through latency indicates how long the barrier takes to process the final arrival and *begin* releasing processors (i.e., the latency from the last arrival at the barrier J_n until the first processor is released R_1). We can therefore define fall-through latency t_{ft} as follows:

$$t_{\text{ft}} \triangleq t_{R_1} - t_{J_n}$$

One goal of a barrier implementation is to minimize this fundamental latency since it reflects time when *no useful work is being done by any processors*. In an ideal barrier, once the final processor J_n arrives at the barrier (satisfying the barrier condition) the processors would immediately be released. In practice, the fall-through latency is a non-zero interval during which the barrier communicates the final arrival, realizes that the condition has been satisfied, and notifies the first processor.

We define fall-through latency to carefully exclude the interval $[J_1, J_n]$ since the arrival times themselves are *not* a characteristic of the barrier, they are a characteristic of the application in which the barrier executes. The nature of the arrival times does have an effect, however, since they may change the contention within the barrier implementation. Fall-through latency is thus not a constant for a particular barrier implementation, but varies. We study two cases for fall-through latency:

The best possible fall-through latency for the barrier primitive is achieved when all the processors but one have arrived at the barrier well in advance of t_{J_n}. In this case, the final arrival finds the least contention within the barrier since every other processor is idling. We call this the "late arrival" case.

On the other hand, the processors might arrive at the barrier at exactly the same time. This case might occur in particular if the application phase is short and there is little opportunity for load imbalance across the processors. In that situation, called the “simultaneous arrival” case, the processors experience contention within the barrier and t_{ft} increases. Each implementation varies in the severity of the performance degradation caused by simultaneous arrivals.

An important goal of any barrier implementation is to avoid performance degradation associated with coincident arrivals. The motivation for this is clear: simultaneous arrivals at the barrier are the *best* achievable load balance in the system! If the barrier operates more slowly in that situation, it detracts from system performance in this otherwise desirable case.

Barrier Release Latency

The second metric of barrier quality, as illustrated in Figure 6.22 is referred to as *release latency*. Release latency corresponds to the latency from when the first processor is released from the barrier R_1 until the last processor is released R_n. Thus, we can define release latency t_{rel} as follows:

$$t_{\text{rel}} \triangleq t_{R_n} - t_{R_1}$$

Unlike fall-through latency, which varies based on the stochastic nature of the arrivals, release latency is to first order a pure characteristic of the barrier itself. In an ideal barrier, release latency is zero, corresponding to all processors being released simultaneously. In practice, the barrier takes a finite time to communicate to the other processors that the barrier has been achieved and release them. In fact, this release phase is subject to contention if implemented inefficiently, which may explode as the number of processors increases.

The second goal of a barrier implementation is to reduce release latency. This is desirable for several reasons. First, releasing processors at different times may introduce a load imbalance in the system at the start of the subsequent phase. A more simultaneous barrier release thus provides better characteristics to the programmer. It may also allow communication in the subsequent phase to be more carefully crafted or provide timing measurements a consistent starting point.

More concretely, release latency reflects the amount of time *wasted* in the barrier primitive’s release phase, and thus release latency should be minimized. For example, if we assume that processors are released uniformly over the interval $[t_{R_0}, t_{R_N}]$, then the average time wasted per processor in the barrier is $t_{\text{ft}} + (t_{R_N} - t_{R_0})/2 = t_{\text{ft}} + t_{\text{rel}}/2$. In general, once the barrier is satisfied, it takes between t_{ft} and $t_{\text{ft}} + t_{\text{rel}}$ for a processor to emerge.

6.2.2 Conventional Barrier Implementations

In this section, we study several indicative barrier implementations on top of shared memory to identify their characteristics and limitations. As in locks, this dissertation does not attempt to exhaustively cover the extremely broad range of software barrier implementations that have been proposed in the literature. Other researchers have studied these alternatives in the context of a variety

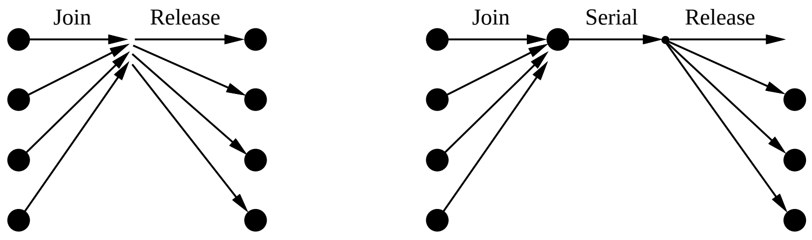


Figure 6.23: Schematic illustration of barrier primitive types: conventional and master-slave. The conventional barrier (left) joins all processors and then releases them all simultaneously. The master-slave barrier (right) joins all processors then releases a single one for a serial/coordination phase. The master then releases the slaves with an explicit command.

of systems [MCS91a, FG91, MCS91b]. Our analysis instead selects a range of primitives that are very common, or illustrate key concepts that are useful in guiding our design. We focus on the MCS Barrier [MCS91a, MCS91b] in particular as a comparison point against FLASH barriers, since it has the highest performance of the conventional algorithms in most cases.

The primary focus of this analysis is on implementations that provide the basic barrier functionality in which all processors rendezvous and then are released simultaneously. A slightly modified barrier primitive is appropriate in some applications such as auto-parallelizing compilers. In the modified barrier, which we refer to as a *master-slave* barrier, the processors rendezvous as usual, except that only a single processor (the "master") is initially released. This allows the master to perform some coordination work in isolation such as setting up parallel computation for the "slaves". Then using an second release routine the master explicitly releases the slaves to complete the barrier. Figure 6.23 illustrates the difference between conventional and master-slave barriers. We present a barrier specially designed for master-slave functionality and also show how other barrier implementations can be converted to a master-slave style.

LL/SC-Based Barrier

We begin as in locks with a simple barrier implementation, which is based on an atomically-updated count of waiters. The pseudo code for this implementation appears in Appendix B, Figure B.3. When each processor arrives at the barrier, it increments the count using LL/SC to guarantee atomicity. Associated with the barrier is also a *generation number* which indicates which barrier instantiation is currently underway. To implement the release portion, a processor reads the current generation number and then spins waiting for it to change. The last processor to arrive resets the count and increments the generation number.

Though it does not contain an explicit lock, this barrier suffers from essentially the same contention problem occurring in LL/SC locks described in Section 6.1.2. Access to the barrier count not only swamps the home node with requests, but also experiences repeated failed SC attempts

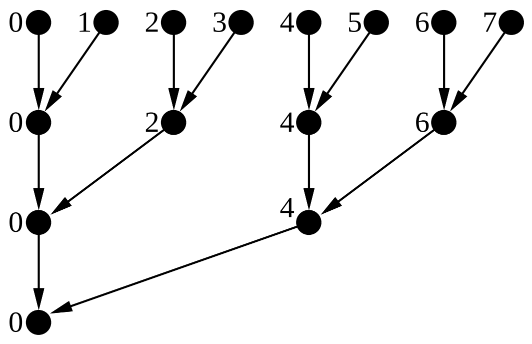


Figure 6.24: Example binary tournament barrier tree for an 8-processor application. In this example, pairs of nodes at each power of two modulus meet in successive rounds, requiring several meetings by some nodes.

if multiple processors attempt the update in close proximity. As a result, the effective traffic to achieve this barrier may grow exponentially in high contention cases (since each additional processor may cause each other processor to fail). In practice, as our microbenchmark results show, this barrier's performance under contention is so poor that it is practically unusable for system sizes of 32 processors or beyond.

Tournament Tree Barrier

As the machine scales, contention for the centralized data structure in the LL/SC barrier implementation skyrockets. A technique proposed by may researchers to address this problem is to implement instead a barrier tree data structure. Hengsen et al. [HFM88] proposed a special tree barrier implementation called a Tournament barrier which uses repeated pairwise synchronization. After every synchronization, one of the two processors proceeds to the next level to synchronize with another processor. This approach, illustrated in Figure 6.24, forms a type of tree in which any meeting experiences contention from only two processors. It can also be generalized to form a tree of any dimensionality or *radix* (i.e., a binary or "2-ary" tree has radix 2). The algorithm we study uses two such trees, one for joining the barrier (shown in Figure 6.24) and another where processors spin to be released (not shown).

Since LL/SC barriers scale so poorly, we use a basic tournament tree barrier implementation as the baseline for our application measurements later in this chapter. The implementation we use was written by Chris Holt as part of the FLASH project. Pseudo code for this implementation appears in Appendix B, Figure B.4. In this implementation, each tree node rendezvous is performed by updating a lock-protected count. By distributing the work across the machine, the contention for any lock is low, but a small degree of artifactual communication from the embedded lock is possible.

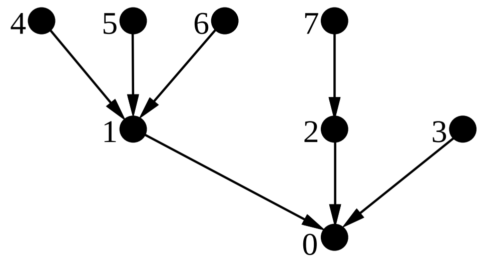


Figure 6.25: Example 3-ary barrier join tree for an 8-processor application.

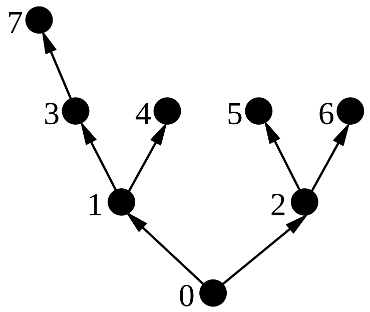


Figure 6.26: Example binary (2-ary) barrier release tree for an 8-processor application.

MCS Tree Barrier

Mellor-Crummey and Scott [MCS91a] present an optimized barrier structure that improves performance still further through a combination of techniques. The pseudo code for the MCS barrier implementation appears in Appendix B, Figure B.5. Unlike most previous barrier approaches, MCS barriers use no locking within their implementation. Instead, each processor is reserved a dedicated flag location to write in its parent's node in the tree. Since there are never multiple writers, locking is not required beyond the atomicity of a single write, which is guaranteed by the processor.

These flags are also intentionally packed together in the parent's structure, which causes false sharing. Our simulations show a counterintuitive benefit from this approach as compared to one where each flag is on its own cache line, even in the simultaneous arrival case. Despite the false sharing it causes during the writes, packing the flags reduces the total number of misses the parent must take to read its children's arrivals. In addition, the cache coherence protocol uses forwarding to transfer lines between false sharing processors, which is fairly efficient.

The MCS barrier uses a different tree structure than the tournament barrier; hypothetical join and release trees of this type are illustrated in Figures 6.25 and 6.26. The benefit of this type of tree is that each node is only required to update a single flag when it arrives, unlike the loop that may walk several tree levels in the tournament-style barrier. Likewise, they also enable significantly simpler release logic, which reduces release latency considerably as compared to the tournament barrier. Note that there is no reason the join and release trees must have the same topology; in this case we show a release tree using a different radix than the join tree.

MCS barriers can also be adapted to provide master-slave functionality merely by returning from the barrier at the root before releasing the children. A second primitive (not shown) can be used to release the children as usual once the master finishes executing the serial computations.

Basic Master-Slave barrier

The third primitive we consider was used in a study of the SUIF auto-parallelizing compiler, and designed specifically to provide master-slave functionality [BAR96]. This primitive is not a tree barrier, but instead is a very simple approach to achieve efficient barriers at small machine sizes. Its pseudo code appears in Appendix B, Figure B.6.

Like MCS barriers, this implementation contains an array of integer flags that are updated independently, avoiding the need for atomic update techniques. It also packs its flag array within the same cache line, causing false sharing. This simple implementation performs very well up to about 8 processors, as do the low-radix tree node joins in MCS barriers. As the number of processors grows, however, performance quickly degrades from the contention for the barrier array.

6.2.3 FLASH Barrier Application Programming Interface

As in locks, we present a library call interface to the barrier primitive, with barriers referenced by a numbered array. Though in practice an application usually needs only one barrier, we support an array in case, for example, several subsets of an application's processors want to use barriers independently.

```
void FLASHBarrier (int barrierNumber, int index);
```

`FLASHBarrier` indicates that the processor with index `index` has reached the given barrier. This primitive returns once the other processors reach the barrier as well. Before this primitive can be used, it must first be configured. As we describe in the next section, this configuration constructs the barrier tree internally between the involved MAGIC chips. We provide a routine that constructs a variable-radix barrier tree:

```
void InitFLASHBarrier (int barrierNumber, int index,
         int totalProcs, int joinRadix, int releaseRadix,
         int masterSlave);
```

This routine constructs the portion of the join and release trees for processor `index` (out of a total of `totalProcs`).[5] It communicates the tree information to MAGIC using a PPC that we describe later in this section. It can generate join and release trees of the same or different radix as specified by the parameters.

Just as in locks, the operating system is required to export an interface to provide the page mappings for access to MAGIC. As before, a memory-based interface allows the barrier resource to be numbered beginning at zero for each application, hiding the sharing of physical machine resources from the user.

[5]Note that we do not specify the actual processor numbers in this interface. Instead we rely on the operating system to provide information about the mapping between application processor indices and physical processor allocation.

Master-Slave FLASH Barrier

FLASH barriers can also provide master-slave barrier functionality. The additional parameter for barrier initialization above, `masterSlave` can be used to select this barrier feature. In this mode, only the root processor (index zero) is released when the barrier is satisfied. Once it has finished the needed coordination processing, it releases the other processors using an additional call:

```
    void FLASHReleaseBarrier (int barrierNumber);
```

Goals of FLASH Barriers

Our barrier protocol is specifically geared to combat the problem of artifactual communication by matching the underlying communication to the barrier operation. This allows the protocol to avoid high contention for shared barrier state and eliminate negative acknowledgements altogether. Furthermore, by eliminating communication artifacts the protocol reduces the impact of simultaneous barrier arrivals, bringing the average performance closer to the “late arrival” best case.

The second goal is to reduce the barrier overhead as reflected in the fall through and release latencies described earlier. Ideally, by reducing overhead we may be able to provide a primitive that performs efficiently over a wide range of machine sizes.

6.2.4 FLASH Barrier Implementation

The FLASH barrier implementation uses some of the same concepts as the FLASH lock primitive introduced earlier in this chapter. Even though barriers seem like a more complicated operation than locks given that all processors are involved, it turns out that the barrier protocol is fairly straightforward.

This occurs for several reasons. First, barrier trees are set up statically, unlike the queues in FLASH locks which are constructed dynamically. Thus, each MAGIC knows in advance its “place” in the barrier. Furthermore, unlike locks, where a centralized resource (the token) is ultimately the focus, barriers lend themselves to parallelism through the tree structure. Thus, no centralized home is needed to direct the operation as was the case with token requests. Finally, there are fewer race conditions in barriers, since the state is very regular.

The resulting barrier protocol achieves excellent performance, contains absolutely no artifactual communication or negative acknowledgements, and is very compact (as we describe in Section 6.2.5).

Barrier Tree Construction and Terminology

Fundamental to the operation of FLASH barriers are two distinct barrier trees. FLASH barriers use a tree structure like that of MCS barriers but the tree structure is stored by MAGIC and is never directly

R	S	T	I	Join Count	Join Total	Rel. Count	Rel. Total	Join Tree Parent	Release Tree Child Array				
									5	4	3	2	1
1	1	1	1	3	3	3	3	8	8	8	8	8	8

Figure 6.27: Barrier protocol internal state format.

visible to applications. As introduced earlier, Figure 6.25 shows a hypothetical barrier join tree. For FLASH barriers, join tree vertices correspond to FLASH nodes while edges correspond to messages carrying requests to join the barrier. Note in particular that interior vertices of the tree coincide with FLASH nodes as do leaf vertices. Before a node can send a message to its parent in the tree, it must have received join requests from all its children as well as the local processor. Leaf vertices each only receive a single join request, from the local processor. Similarly, Figure 6.26 illustrates a release tree for the same nodes. Here the edges correspond to release messages indicating the barrier has been satisfied.

To represent these trees internally, each node in MAGIC maintains an 8-byte record for each barrier. Figure 6.27 shows the internal MAGIC barrier protocol state format used to store this information. The barrier configuration in this state is stored statically and is provided to the local MAGIC by a PPC performed within the barrier initialization call. The version we show uses 8-bit pointers and can thus support systems up to 256 processors. Larger systems can be supported by reducing the number of children permitted and increasing the pointer size. The fields are used as follows:

Root (R) Set if this node is the root of the barrier tree.

Sense (S) A toggling sense bit indicator used in reporting barrier completion to the processor. This is used to avoid a race condition in processor-MAGIC communication and improve performance.

Two Phase (T) Indicates whether the barrier is configured for two-phase operation, which causes release behavior to differ at the tree root.

Initialized (I) Indicates the barrier has been configured and is ready for use. Access to an uninitialized barrier should deliver an exception to the application via the operating system.

Join Count This count tracks the *current* number of processors which have arrived at the barrier. This consists of messages from the network as well as join indications from the local processor.

Join Total This read-only field stores the total number of joins expected at this node; when that many is received the vertex can notify its parent. The value in this field is larger than the Release Total because it counts the local processor join as well.

Release Count When a barrier release has only partially completed due to outgoing network queue limitations, this field stores the number of children which have been notified. By separating join and release, early-released processors can join again immediately without affecting the release.

Release Total This read-only field stores the number of release tree children for this node, which corresponds to the number of valid children in the release tree child array.

Join Tree Parent Indicates the barrier join tree parent node, unused for the root node (we detect the root using the R bit for higher performance).

Release Tree Children A packed bit vector of release tree children, the nodes which must receive a release indication when the barrier is achieved.

Though our approach benefits all application sizes, it is especially suited for large-scale parallel applications, since those are the ones which tend to use barriers the most, and tend to suffer the most from the artifactual communication in software techniques. For performance, those applications also typically run with processes attached to a particular processor, thus our choice of static barrier trees is appropriate. However, due to the static tree the barrier must be reconfigured should the process/processor mapping change due to machine scheduling or multiprogramming (this function could be performed transparently to the application by the operating system communicating with MAGIC).

A related effect arises from parallel applications running in degenerate modes such as with multiple processes on single processor. This situation is supported by the protocol, but the tree must be configured that way in advance by incrementing the Join Count field. Thus it merely appears as if multiple barrier participants are joining on the local node. Here, as in the case above, multiprogramming effects which cause the processor mapping to change require the barriers be reconfigured.

Barrier Join

The join phase of the barrier begins when the first processor calls `FLASHBarrier`. The barrier join is similar to the token request in locks, but with a small difference. In locks, the cache is checked first to see if the token is present before it is requested. In barriers, no tokens are used—instead, the processor immediately issues a PPR to MAGIC asking to join the barrier. This PPR is analogous to the token request in locks: since it is uncached, the barrier join operation is reliable and atomic. The join PPR is a read, and it returns the value of the barrier sense indicator *before* the barrier. The sense indicator is a single bit that toggles when the barrier is achieved.

The library call then requests the cacheable barrier state, which is provided by the barrier protocol and not the cache coherence protocol (similar to the way FLASH lock state requests are handled).

For barriers, this state contains merely a single valid word that is the current value of the barrier sense indicator. The barrier spins on this value until it is the opposite of the starting sense returned from the initial PPR. By returning the starting sense indicator value from the PPR we avoid a race in which the barrier is satisfied after the PPR but before the barrier status is requested (this might occur if the processor is the very last to arrive). By relying on the starting sense value from the PPR, the processor immediately detects the barrier as satisfied when the cacheable state returns.

Within MAGIC, the join request cues the barrier tree to be consulted. If the join represents the last join expected at this vertex of the join tree MAGIC sends a join message to its parent in the tree. This carries on until ultimately the final join arrives at the root, cueing the root node to begin the release phase, described below.

The protocol code for the join phase is very compact. The only exceptional case arises when a join arrives in the network that cues a message to the next level of the tree. If network queue space is not available for the outgoing join message the software queue is needed to issue the message later. Fortunately, the software queue retry always succeeds since the SWQ is selected only when queue space is available.

Barrier Release

When the final join reaches the root, MAGIC begins the release phase. Each release event causes three actions to occur:

- It updates the barrier state to reflect the release. This includes toggling the sense indicator both to indicate the barrier is satisfied and to prepare for the next barrier. It also initializes the join counts so that the barrier operates correctly in case some processors join the barrier again before all others have been released.
- It invalidates the local processor's cached barrier state to indicate the sense has changed.
- It sends messages to its children in the release tree, indicating they should release in turn.

Meanwhile, the processor barrier routine is spinning on the sense bit in the barrier state. When it receives the invalidation, it requests the state again, detects that the barrier sense bit has changed, and exits the barrier routine allowing the application to resume.

Within MAGIC the release phase has similar network deadlock concerns as does the join, in fact slightly more complex. When a release message arrives at a node, it often must send *multiple* release messages out to the next level of the tree. The software queue is needed here as well and in fact several retries may be needed depending on the release tree radix.

Master-Slave Barrier Variant

Supporting the master-slave barrier variant [BAR96] requires only a small modification to the protocol. The root node normally detects the barrier is complete and then launches the release. For barriers marked as master-slave, the satisfied barrier notifies only the root processor (by invalidating its cache) and then stops. By requiring the master processor to be the root of the barrier trees, releasing the master alone is straightforward.

The master then detects in the standard way that the barrier has been satisfied and can perform coordination processing. Once it finishes, its call to `FLASHReleaseBarrier` cues the protocol to begin the release phase by notifying the children in the release tree.

6.2.5 Protocol Handlers

As in locks, the barrier protocol is very compact, allowing it to work in tandem with cache coherence without degrading performance significantly from cache effects. Table 6.4 enumerates the handler components of the protocol, which is slightly larger than 2 KB in total. The compactness in this protocol arises from many of the same characteristics as in the lock protocol, described in Section 6.1, including the use of a special coherence protocol for barrier state accesses.

6.2.6 Barrier Performance in Isolation

In this section we evaluate the performance of the FLASH barrier protocol using the metrics introduced in Section 6.2.1. We study barrier primitives in the context of microbenchmarks that isolate the two extremes of barrier performance: late arrival and simultaneous arrival. We also study the range of barrier primitives introduced earlier, to compare FLASH barriers against existing techniques. Finally, we briefly consider master-slave barrier primitives. In that case we consider the basic master-slave barrier as well as a master-slave variant of the MCS barrier. We describe several master-slave implementations using FLASH barriers, highlighting the benefit that flexibility provides for allowing protocol customization. Later, in Section 6.3, we consider the performance impact of FLASH barriers in the context of SPLASH-2 applications. This section uses the same simulation environment that was used for locks.

Fall-through Latency

We begin our analysis with barrier fall-through latency (t_{ft}) for the range of primitives we study. Recall that fall-through latency corresponds to the latency from the last arrival at the barrier until the first processor is released. This latency is not constant but instead varies based on the processor arrival characteristics. The best performance occurs in the late arrival case in which all but one processor is waiting at the barrier and the final processor arrives much later. Table 6.5 presents

Table 6.4: Summary of the FLASH barrier protocol handlers.

Handler Name/ Handler Size (bytes)	Description
BarrierStatusRead 112 B	The "cache miss" handler for the barrier coherence protocol. This provides the processor with the current barrier sense indicator value, that toggles when the barrier is achieved.
BarrierReplacementHint 16 B	This handles barrier state being replaced from the cache. Unlike locks, the barrier protocol does not need to track the caching of barrier state, so this handler is merely a nop.
PIBarrierJoin 568 B	Accepts a processor request to join the barrier. This is invoked explicitly by the barrier join PPR.
PIBarrierRelease 232 B	For master-slave barriers, `FLASHReleaseBarrier` causes this handler to release the other waiters once the master's coordination processing is complete.
NIBarrierJoin 552 B	Accepts a join request from the network. If all nodes have joined it passes the join to the next level of the tree, or begins the release phase (if the join has reached the root).
NIBarrierRelease 272 B	Accepts release requests and propagates it to the children in the release tree.
ReleaseBarrier 176 B	A shared subroutine that sends release requests to the children in the tree. Shared by NIBarrierRelease and SWBarrierRelease.
SWBarrierJoin (88 B) SWBarrierRelease (224 B)	Software Queue handlers to resume a join or release suspended by queue limitations.
Total size: 2240 B	

the late arrival fall-through latency for the barrier primitives we study, using the root node as the final processor to arrive. At the other extreme, simultaneous arrival, every processor arrives at the barrier at exactly the same time. In this case, contention for barrier resources is at its greatest and performance decreases as a result. Table 6.6 presents the fall-through latency for simultaneous arrival.

In practice, the barrier performance falls somewhere between the two extremes of late and simultaneous arrivals. Moreover, these tests were carried out in carefully-crafted isolation to eliminate other activity in the system. Within an application, the current activity in the system such as that from the cache coherence protocol may affect the barrier performance in ways other than those reflected in these tests.

For late arrival, LL/SC performs well up to 16 processors, but for larger sizes and for even small systems under simultaneous arrival its fall-through latency explodes from contention for the barrier data structure. This contention clearly motivates the need for tree barriers for even mild-sized systems.

Table 6.5: Barrier fall-through latency (t_{ft}) for late arrival, using a range of primitives and parameters (microseconds). For consistency, the tree root processor is the last one to arrive at the barrier.

Total Procs.	LL/SC	Tourn 2-ary	Tourn 4-ary	MCS 2-ary	MCS 4-2-ary	MCS 4-ary	FLASH 1-ary	FLASH 2-ary	FLASH 3-ary	FLASH 4-ary	FLASH 5-ary
4	3.6	4.9	4.8	2.5	2.6	3.7	1.4	1.4	1.4	1.4	1.4
8	3.6	4.8	4.8	2.5	2.5	5.3	1.3	1.4	1.4	1.5	1.5
16	3.7	4.9	4.9	2.5	2.6	5.7	1.3	1.4	1.4	1.5	1.5
32	73.5	5.0	4.9	2.6	2.5	5.9	1.3	1.4	1.4	1.5	1.5
64	186.8	5.1	5.0	2.6	2.6	5.9	1.3	1.4	1.4	1.5	1.5
128	771.6	5.2	5.1	2.6	2.5	5.8	1.3	1.4	1.4	1.5	1.5

Table 6.6: Barrier fall-through latency (t_{ft}) for simultaneous arrival, using a range of primitives and parameters (microseconds).

Total Procs.	LL/SC	Tourn 2-ary	Tourn 4-ary	MCS 2-ary	MCS 4-2-ary	MCS 4-ary	FLASH 1-ary	FLASH 2-ary	FLASH 3-ary	FLASH 4-ary	FLASH 5-ary
4	8.4	10.4	15.3	6.7	7.1	8.4	2.6	2.1	2.0	2.0	2.0
8	25.8	15.0	18.8	10.8	11.2	14.0	4.9	2.7	2.6	2.6	2.5
16	70.9	18.3	28.4	15.9	16.1	18.6	9.5	3.3	3.2	2.9	3.1
32	171.4	21.9	32.6	20.1	19.3	22.6	18.7	3.9	3.7	3.5	3.0
64	419.6	25.7	40.8	26.2	26.0	29.3	37.3	4.6	4.5	4.0	4.2
128	1000.2	31.6	45.7	32.9	30.3	34.6	74.4	5.5	4.6	4.7	4.4

The first tree technique, the tournament tree barrier ("Tourn"), scales much better though it is essentially the slowest absolute performer of all the tree primitives. For late arrivals, the tournament tree structure easily explains the result, since the last arrival must traverse the several rounds of the tournament even if it is the root. For simultaneous arrivals, contention for the lock at each round causes the performance degradation as compared to late arrival. Note especially that the 4-ary tree performs slightly better for late arrival since it reduces the depth of the tree, while the 2-ary is superior for simultaneous arrival since contention is higher when barrier rounds involve 4 processors.

The MCS barrier scales similarly to the tournament barrier, though its absolute performance is slightly better. For MCS we show not only 2-ary and 4-ary join trees but also a hybrid barrier using radix 4 for join and radix 2 for release, referred to as MCS 4-2-ary. This additional variant allows us to isolate several interesting effects.

For late arrivals, the traditional tree structure used by MCS allows the tree root to join with only a single round, unlike the tournament approach. This accounts for the difference between the 2-ary fall-through latency for MCS and Tourn. Aside from the number of rounds, by comparing the 4-processor late arrival result for a 2-ary Tourn and 4-2-ary MCS (each of which has only one round), we see that the pre-computed tree of MCS is also more efficient than the dynamic tree calculations performed in the tournament barrier.

Finally we present the FLASH barrier, which for tree radix of 2 or more outperforms every other technique at every machine size. We present FLASH barriers with tree radix 1 (corresponding to a line of processors, a degenerate tree) to illustrate that the communication within the primitive is very efficient. Up to 32 processors, even this naive barrier configuration outperforms the MCS barrier. It also shows the benefit provided by a tree for simultaneous arrivals. Though increasing FLASH tree radix provides essentially monotonic improvement, the bulk of the gains is achieved merely by radix 2.

In the simultaneous arrival case, the benefits of FLASH barriers are particularly evident. Under the contention simultaneous arrivals cause, every other technique incurs some form of artifactual communication, the impact of which increases with machine size. By matching the inherent communication needed by a barrier, the FLASH technique eliminates these artifacts and achieves nearly an order of magnitude speedup at 128 processors as compared to the next best technique, the MCS barrier.

The results also show an interesting result that is not immediately obvious: late arrival t_{ft} for FLASH barriers *increases* with increasing radix. This effect can also be seen in the difference between late arrival fall-through latency in MCS 4-2-ary and MCS 4-ary. Even though t_{ft} measures only the initial release from the barrier, this effect occurs from the increase in *release* tree radix. In a tree barrier, the root is the first processor released, so the fall-through latency generally reflects the time for the root to emerge. When the radix grows, however, the root must perform more work (i.e., release more processors) before leaving the barrier primitive. In the case of MCS, this work happens inline in the barrier software, and thus the increase in late-arrival t_{ft} between MCS 4-2-ary and MCS 4-ary is noticeable, 2.8μs at 128 processors. The same effect occurs in FLASH barriers but in a different way. Unlike MCS, once the barrier is achieved MAGIC notifies the tree root processor *immediately*. However, before that processor can leave the barrier, it must fetch the status word and view the barrier sense change. That miss cannot be satisfied by MAGIC until the barrier protocol handler finishes releasing the root's children. Fortunately, in FLASH unlike MCS the difference is very small, only 0.1μs at 128 processors.

Release Latency

The barrier release latency results are very straightforward to analyze, and are completely independent of the nature of the barrier arrivals. We therefore present a single set of release latency results, shown in Table 6.7, that applies to both late and simultaneous arrivals.

LL/SC barrier release is inefficient, as expected, due to the rush of requesters to read the barrier generation number when updated by the home. Surprisingly, the release phase in tournament tree

Table 6.7: Barrier release latency (t_{rel}) for a range of primitives and parameters (microseconds).

Total Procs.	LL/SC	Tourn 2-ary	Tourn 4-ary	MCS 2-ary	MCS 4-2-ary	MCS 4-ary	FLASH 1-ary	FLASH 2-ary	FLASH 3-ary	FLASH 4-ary	FLASH 5-ary
4	3.2	5.9	5.4	4.8	4.8	3.4	1.2	0.8	0.2	0.2	0.2
8	7.4	14.5	16.7	8.4	8.4	5.5	3.4	1.4	0.9	0.9	0.9
16	15.4	25.8	22.3	12.0	12.0	8.9	7.7	2.0	1.6	1.0	1.0
32	31.7	39.8	41.9	15.6	15.7	12.0	16.2	2.6	1.6	1.8	1.7
64	63.6	57.1	47.8	19.7	19.7	16.3	33.6	3.3	2.3	1.9	1.9
128	127.0	78.1	77.7	24.5	24.2	21.4	68.6	4.0	3.1	2.8	2.8

barriers is also fairly inefficient, despite its use of a release tree. This occurs for several reasons, some of which are due to the implementation we were given and not fundamental to the tournament algorithm in general. First, during the release, it employs lock-protected flag update just as in the join, even though each release tree flag has only one writer. This overly conservative approach causes extra references in the release critical path. Second, as in the fall-through case, the tournament-style barrier means each processor loops and may need to release processors at several levels in the tree. Furthermore, it orders the processor releases inefficiently, causing the release tree propagation to be nonuniform and reducing the parallelism it might otherwise achieve.

MCS barriers perform much better, due in part to the very simple release structure that allows much lower overhead than the tournament tree barrier. As for the choice of release tree radix, the original description of MCS barriers [MCS91b] suggests that a release tree of radix 2 is best based on a theoretical analysis. In contrast, our results show a MCS having consistent though mild improvement from increasing release tree radix from 2 to 4. The difficulty of predicting real machine behavior, especially due to contention, argues for a execution- or simulation-driven methodology to select parameters rather than a purely theoretical one.

FLASH barriers once again outperform the other techniques by a wide margin. In the release phase of a FLASH barrier, the release messages are delivered directly to the nodes in the tree. When they arrive, the barrier protocol on MAGIC then propagates those release messages to other nodes in the tree without any processor interaction. The result of this design is that *no remote cache misses* are taken by any processor to determine the barrier has been satisfied. Instead, each processor's local barrier status flag is invalidated at the same time the barrier's state is modified in MAGIC to indicate completion. All that is required is a local cache miss for the barrier state and the processor is released.

Contrast this with even the most efficient software tree release technique. Using the processor, even updating a simple flag involves acquiring ownership and invalidating the currently spinning processor(s). Then in response the previously spinning node must miss remotely to fetch the updated value. Only then can the waiter propagate the release to its children in the barrier tree in the same manner. Avoiding these repeated processor cache misses in the critical path and encapsulating this entire release tree process in MAGIC is one of the most clear-cut successes of the FLASH barrier protocol.

Master-Slave Barrier Performance

We now turn to the master-slave barrier variant to consider the performance of various primitives in that different mode. We study several classes of conventional primitives, beginning with the basic master-slave primitive [BAR96], and a two-phase variant of the MCS barrier. We also study three ways in which master-slave barriers can be implemented with FLASH barriers.

In the first of these implementations, we observe that conventional FLASH barriers performed so well in our previous tests that one viable approach may be to use *two* back-to-back FLASH barriers, with the serial portion occurring in the middle. We refer to this as the "FLASH Naive" approach. Our second implementation, referred to as "FLASH+Flag", uses a FLASH barrier followed by a spin loop on a shared memory flag. The flag is set by the master after it completes the serial computations. Finally we consider the custom designed master-slave version of FLASH barriers described in Section 6.2.4 in which the master alone is released once the barrier is satisfied. After its serial processing the master performs an additional PPR to cue the release. The advantage of this approach is that it uses the optimized tree release mechanism. For comparison purposes and to help ascertain the overhead of the extra processor-MAGIC communication, we also show the results of the same benchmark for a conventional FLASH barrier *not providing* master-slave functionality but merely releasing all processors simultaneously.

For this evaluation we use a microbenchmark that simulates the operation that takes place within an automatically parallelized application. In a loop iterating 32 times, we simulate the processors joining at a master-slave barrier. The master processor is released first, and then immediately releases the slaves. By eliminating the actual parallel and serial work in this benchmark it focuses exclusively on synchronization performance and illustrates the overhead from barrier synchronization in the limit as the computation becomes very fine. The latency we report for this benchmark is the average length of one iteration of the loop, which corresponds essentially to the total overhead incurred from the barrier primitive.

Table 6.8 shows the results over a range of machine sizes of interest for this style of primitive. The basic master slave barrier performs very well for small machines, even surpassing MCS up to 8 processors. Beyond that, however, contention for the barrier renders it unusable. MCS barriers were easily adapted to master-slave mode and show similar scaling to the conventional mode described previously.

The FLASH barrier variants show several useful results. First and foremost, even the Naive solution using two back to back barriers outperforms the conventional implementations. Note that

Table 6.8: Master-Slave Barrier microbenchmark results, showing the average iteration latency of back-to-back master-slave barriers with immediate release by the master. The latency for a conventional 4-ary FLASH barrier not providing master-slave functionality is shown on the right for comparison purposes.

Total Processors	Master-Slave Barrier Implementations: Basic M-S	MCS M-S 2-ary	FLASH Naive 4-ary	FLASH + Flag 4-ary	FLASH Custom 4-ary	FLASH Conv. 4-ary
2	3.9	4.6	3.9	3.3	2.5	2.0
4	9.2	10.3	4.9	5.2	3.0	2.5
8	19.9	21.7	7.6	9.2	4.3	3.8
16	43.0	24.4	8.3	18.8	4.7	4.2
32	286.6	36.7	11.2	35.1	6.1	5.6

the Naive approach latency is (within error margins) exactly double that of the conventional (non-master-slave) FLASH barrier. As might be expected, the FLASH+Flag approach does not improve performance due to contention for the flag, illustrating importance of using a tree-based release mechanism. The best performer overall is the custom designed FLASH master-slave variant. It scales in essentially the same way as the conventional FLASH primitive on the right, but slows uniformly by 0.5 μs from the latency to notify the processor and receive the PPR that releases the slaves.

6.2.7 Related Work

In Section 6.2.2 we presented several key barriers as background for our protocol design and implementation. Below we describe the broader background research on barrier synchronization.

A novel barrier implementation called the butterfly barrier is described by Brooks [BI86] in which a series of pairwise synchronizations is carefully orchestrated between the processors. Once this $O(\log n)$ series completes, the processors are guaranteed to have arrived at the barrier. Though each synchronization is based on locks, by distributing communication the butterfly reduces contention for each lock to only two processors. Similar logarithmic performance is provided by tree-based barriers, and many implementations including FLASH barriers also reduce the total number of communications in the critical path as compared to the butterfly.

Hengsen et al. [HFM88] make three modifications to the butterfly barrier that successively improve it. First, they introduce a sense change to reduce the barrier code size slightly. Next they reduce barrier traffic by a factor of two through a *dissemination* algorithm, in which the symmetric pairwise synchronizations are replaced by an asymmetric communication pattern that also satisfies the condition. Finally they present yet another modified communication pattern they call a *tournament*, which we described in Section 6.2.2. In the original tournament barrier, all processors ultimately spin on a single flag, which is a source of contention in invalidation-based cache coherence protocols.

In their journal paper on synchronization, Mellor-Crummey and Scott [MCS91a] present a survey of the major classes of barrier implementations. They first optimize the tournament barrier of Hengsen et al. through the use of a release tree, thus eliminating the contention for a single flag. They also introduce a new barrier primitive that improves performance still further through careful data structure and code optimizations [MCS91b], which is the MCS barrier we describe in Section 6.2.2.

Similar to Mellor-Crummey and Scott, Woest and Goodman [WG91] present a survey of barrier techniques, but with an emphasis on their implementation using the QOLB primitive we describe in Section 6.1.9. In that aspect they find the QOLB implementations to have no significant advantage in particular because barriers do not utilize the collocation benefit QOLB provides.

Woest and Goodman also study the barrier release phase, including the use of a broadcast (update) write primitive to quickly update all waiters. Interestingly, they find that even though the broadcast write release is extremely efficient, the initial serialized read misses to fetch the notification address to each processor must be amortized over many barriers for the combination to show gains. Only a combination of broadcast write and combining reads would make the approach tractable; as a result, they conclude that tree wakeup may be a superior approach in general [WG91].

In a similar approach to that used by Anderson for locks [And89], Agarwal and Cherian [AC89] study various software back-off strategies to reduce contention in barriers. One major focus of their analysis was reducing the exploding network traffic arising from barriers under contention, as well as decreasing overall waiting time. They present several back-off implementations and parameters which achieve a balance between these two goals, though their focus is on non-tree barriers. The barriers they study are not appropriate for large scale systems, however, and tree barrier implementations eliminate much of the need for the back-off techniques they study.

The Cray T3D [Cra93] provides a dedicated physical barrier/eureka[6] network that can synchronize the entire machine in less than 2 μs. The T3E [Sco96] virtualizes the barrier support to make it easier to share by providing 32 different barrier/eureka units at each processor, while maintaining approximately the same performance. T3E also enables "fuzzy" barriers in which the processor can perform unrelated work while it waits at the barrier, then either poll or be interrupted to detect the barrier is achieved. FLASH barriers can provide fuzzy semantics merely by extending the API with a call that returns after issuing the join indication to MAGIC and one which polls the current barrier status.

[6] A "eureka" is a non-blocking parallel-OR function, enabling a processor to quickly indicate completion to the others, such as in a parallel search implementation. By comparison, a barrier is a blocking parallel AND.

Scalable Counting Techniques

Many researchers study scalable counting techniques, one application of which is the implementation of barriers. In [HLS95], Herlihy et al. compare counting implementations using a range of hardware and software support. They find that lock-based techniques degrade quickly as machine size increases and that only software combining trees or counting networks were truly scalable. They show software-only implementations of counting trees and counting networks using both shared memory and message passing techniques. In their experiments on the MIT Alewife, both primitives scaled well, though the message passing implementations achieved approximately twice the throughput. The FLASH barrier implementation uses messages between MAGIC chips in a similar way to the primitives they describe.

Freudenthal [FG91] describes the hardware-supported scalable counting technique Fetch-And-Add to provide barriers and other types of synchronization. We could mirror these techniques by providing Fetch-and-Add using protocol support in MAGIC (as we describe in Section 7.2.1) and then in turn building barriers on Fetch-and-Add. Still, the FLASH barrier protocol we describe achieves much higher performance by instead tailoring the communication specifically for barriers.

6.3 Lock and Barrier Performance Impact in Applications

This section considers the performance benefits of FLASH locks and barriers within the context of real applications from the SPLASH-2 benchmark suite. We show three applications, to illustrate a number of different ways that synchronization is used in practice, and to study whether or not synchronization improvements translate to application gains.

Application analysis of synchronization is significantly more difficult than the microbenchmark approach used earlier since many effects occur at once. We use the intuition and the baseline performance metrics provided by the microbenchmarks as an overall guide to our analysis. However, it is important to note that the performance metric results we measured using microbenchmarks are usually not attainable in applications due to other interfering effects. Furthermore, the microbenchmarks cannot predict all the performance gains or losses we encounter, since synchronization interacts with application communication, unlike the controlled environment of the microbenchmarks. In particular, at large machine sizes we sometimes encounter gains larger than the microbenchmarks predict. These arise because shared memory contention is reduced by the use of optimized synchronization, and thus unrelated shared memory accesses improve as well. This effect is particularly prevalent in FLASH, which relies on negative acknowledgements to avoid deadlocks when queues fill. As a result, FLASH encounters “cliffs” where high contention causes traffic to increase further as NAKs are generated.

For the purposes of this study, we broadly classify poor scalability into two major categories. In some cases, fundamental contention for shared resources, load imbalance, or other effects limit application parallel efficiency and prevent additional processors from reducing execution time. In other cases, the application is designed suitably and is inherently capable of scaling well, but inefficient synchronization limits the performance. We focus on this latter case, to determine whether optimized synchronization can improve efficiency and scalability. Restructuring the application algorithms to address the more fundamental scalability limits of the former case is beyond the scope of this dissertation.

6.3.1 Application Descriptions and Characteristics

To study this problem, we have selected three scientific applications, all from the SPLASH-2 benchmark suite [WOT+95]. The applications we consider are Water, a water molecule simulation, Barnes, an N-body simulation based on the Barnes-Hut method, and Ocean, a simulation of ocean currents. These applications are fairly well suited for the synchronization primitives we study since they use significant locks or barriers and scale well to reasonably large machines (32 processors or larger) where our techniques become applicable. The other SPLASH-2 applications we do not study may benefit from our primitives as well, though their use of synchronization tends to be more mild. For our scalability measurement we consider *constant problem size* scaling—the problem size is held fixed while the machine size is increased. As such, we focus on computation time as our primary metric, which highlights concretely the benefits from adding additional processors. We study relatively small problems for these applications both for simulation tractability and to explore the scalability constraints imposed by conventional synchronization techniques. It is important to realize that real systems may not execute applications with this high degree of synchronization, but we use them to stress the synchronization mechanisms intensely and push the limits of the primitives.

Our study focuses exclusively on the synchronization characteristics in these applications; other researchers have studied their behavior more generally. Woo et al. [WOT+95, Woo96] present their structure and design, and describe their working set and communication characteristics in detail. Kuskin [Kus97] and Heinrich et al. [HKO+94] present simulation results for Barnes and Ocean and describe their overall behavior on FLASH, focusing in particular on the effects of flexibility on memory access times. Table 6.9 summarizes the synchronization within these applications; the exact counts vary based on machine size. We begin by describing the applications and identifying the nature of their synchronization usage.

Water

Water is a molecular dynamics application that evaluates the forces and potentials that occur over time in a system of liquid water molecules. The algorithm computes the Newtonian equations for

Table 6.9: Application problem size and synchronization usage overview. Synchronization usage counts vary by problem size and/or processor count.

Application	Problem Size	Processors	Locks	Barriers
Water	512 molecules	16–128 procs	19,000–135,000	17
Water	1024 molecules	32–128 procs	70,000–273,000	17
Barnes	8192 bodies	16–128 procs	8,000–10,000	16
Ocean	256x256 ocean	16–64 procs	400–1,600	143
Ocean	514x514 ocean	32–64 procs	500–1,200	156

motion of water molecules within a box, iterating over a number of time steps in an attempt to reach steady state [SWG92].

We consider the "N squared" version of Water presented in the SPLASH-2 suite, which uses an $O(n^2)$ algorithm to compute the inter-molecule interactions. The extensive use of locking in Water primarily arises in this phase. Each processor first computes in a private array the change in inter-molecular forces arising from its molecules. Then it applies those changes to other processors' molecules under lock protection per molecule (or group of molecules). Water uses several barriers to divide phases, but as Table 6.9 illustrates, its synchronization is overwhelmingly dominated by locks, so we focus on its locking characteristics.

Barnes

Barnes simulates the interaction of a set of bodies in three dimensions over a series of time steps. It uses the Barnes-Hut hierarchical N-body method to compute the interaction forces between the particles [HS95]. The SPLASH-2 implementation of Barnes includes a number of data structure optimizations to improve performance [WOT$^+$95]. The particles are stored in an octree structure, which is traversed repeatedly (once per particle) to determine the force on each particle. In a separate phase, the forces are then used to update each particle's position, and the process iterates.

The complex structure of Barnes causes it to enlist a variety of synchronization primitives. Most notably, Barnes uses significant locking, in particular an array of locks used in two different portions of the algorithm. First, each group of bodies is protected by one of these locks as the program walks the tree to update both forces on a body and their position. Locks are also used to coordinate during a load balancing phase in which the assignment of bodies to processors is adjusted.

Barnes also uses a flag per tree element to determine when the tree walking has finished and the processor can enter the barrier. These updates only have a single reader and writer, and thus do not require mutual exclusion through locks. Like Water, Barnes also uses barriers, though the phases in Barnes are very long and thus the barriers account for only a trivial portion of the runtime. Therefore, we focus our analysis of Barnes on its locking characteristics.

Ocean

The Ocean application models large-scale ocean movements over time based on eddy and boundary currents. We use a recent version of Ocean containing modifications to improve performance on shared memory systems, part of the SPLASH-2 benchmark suite [WOT$^+$95]. In particular, the ocean model is partitioned using a 4-D grid, so that grid portions can be allocated on the processor which manipulates them. This version also applies a red-black Gauss-Seidel multigrid solver [Bra77] in place of the SOR solver used in prior versions. Finally, it includes extensive prefetching instructions to improve the performance of shared memory accesses.

Ocean is based on an iterative algorithm, and each iteration is further broken up into ten phases. As a result, Ocean makes extensive use of barriers to protect the consumption of data generated in the previous phase by other processors, executing approximately 150 barriers overall. In fact, only a very small amount of locking is done in Ocean, thus the potential for gains from optimized locking is small. We focus our analysis of Ocean instead on its barrier synchronization.

6.3.2 Water

The baseline version of Water is based on LL/SC locks, just as in our microbenchmarks. We generate two other versions, one using MCS locks, the other using the FLASH locks presented in Section 6.1.[7] As before, LL/SC and MCS locks each use exponential back-off when contention is detected in their atomic updates. In the environment we use, we are able to simulate systems up to 128 processors before encountering limits in our simulators themselves and the simulation hosts. This limitation is not fundamental to FLASH or the protocols we describe. In particular, we expect the potential for gains from FLASH locks only to increase at larger machine sizes.

We use an array of 512 locks at all problem sizes to protect the molecule data structures. We configure all the implementations to match this limit so the results can be compared directly. For 512 molecules, this allows a single lock per molecule while at the 1024 molecule problem size, two molecules share a lock. In practice, locks would not be shared in this manner due to the contention it causes. We include these results merely to illustrate what might happen if much higher contention is encountered in an application that poorly structures its synchronization.

Synchronization Usage

Water uses locks to protect updates to shared data structures in two different ways. First, it employs individual locks to guard updates to several central summary variables. More importantly, it uses an array of locks associated with a molecule (or a group of molecules) in the problem, which protect

[7]We study the use of the normal FLASH primitive, not the Aggressive variant described in Section 6.1.8.

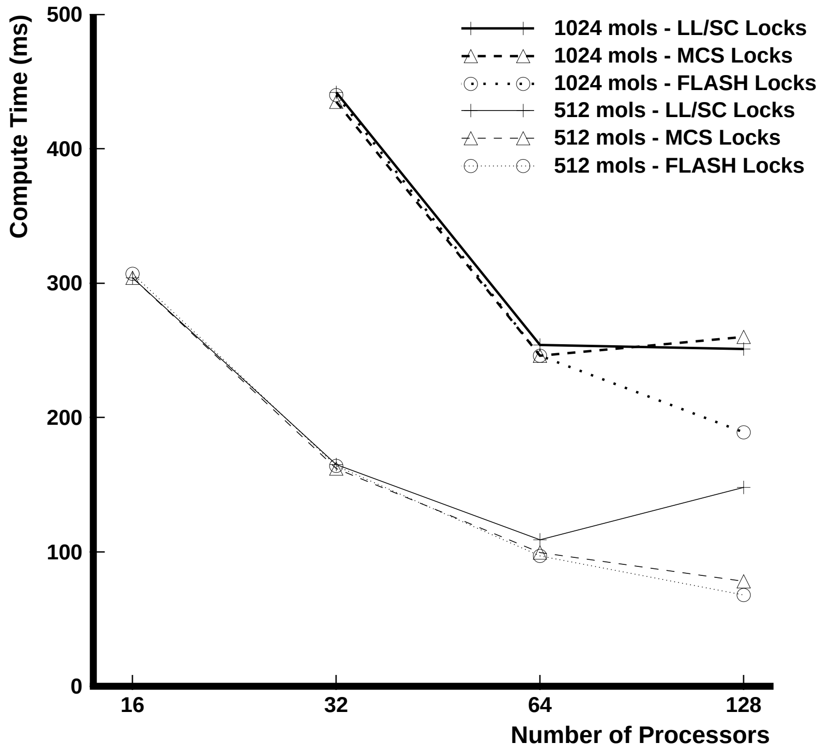


Figure 6.28: Water computation time using different lock techniques.

updates to the current force on each molecule. Our three application variants for Water convert both uses of locks to use the primitive under study (LL/SC, MCS, or FLASH).

In the normal SPLASH-2 version, the array of locks is typically padded to avoid false sharing, but is allocated from memory on a single node. Our initial simulations found that this centralized allocation was a serious problem in the application since the node where the locks are allocated is overwhelmed with requests and becomes a bottleneck. To eliminate this problem and allow LL/SC and MCS locks to perform as well as possible, we modified the lock array to distribute the lock allocation across the nodes of the machine and stripe the lock ordering to reduce contention as much as possible. For FLASH locks, which can use any node as the home, we matched this distribution exactly to attempt to reproduce the same behavior. We do not present the results for the poorly allocated application, but the gains from distribution were significant: LL/SC lock compute time for 512 molecules decreased by 32% at 64 processors, 67% at 128 processors. MCS locks saw similar gains as well.

Simulation Results Format

We run Water at two problem sizes: 512 molecules and 1024 molecules, and over machine sizes from 16–128 processors, excluding the smallest machine for the larger problem size. The computation time for Water (execution time excluding initialization) is presented in Figure 6.28.

To analyze the application behavior in more detail, we present a table of detailed statistics about each simulation and a series of histograms that illustrate the lock metrics introduced earlier. We begin by explaining the format of each, since the data for other applications is presented similarly, then return to Water specifically and describe its table and histogram results.

Table 6.10 presents the resulting statistics for Water, with one row for each simulation. Each group of three rows corresponds to a particular problem and machine size, explored using the three different locking primitives we study. The second column presents the computation times illustrated in Figure 6.28. The next two columns show the percentage of time the application spent in synchronization, expressed as average and maximum times over the processors. This statistic is important, but it is not always straightforward to use for this analysis since it does not differentiate fundamental wait time due to inherent mutual exclusion in the application from the performance of the primitive itself. To complement the sync time and help separate these effects, the far right columns show the lock metrics we introduced earlier, measured using our simulator's ability to watch for synchronization primitives without perturbing the application. Those columns present the acquire latency and hand-off latency metrics as before, as well as a breakdown showing the fraction of locks encountering the acquire case (available when requested) or hand-off case (held when requested).

In many cases we find that contention for the protocol processor itself can be significant in determining the performance of the application. The middle columns attempt to characterize this contention in two ways. First, we show the average read miss latency encountered by the application. As an aggregate statistic, this must be used with caution, but it generally provides a good overall indication of the effects of contention. Next to it we show protocol processor occupancy, i.e., the fraction of time the protocol processor is busy. Like sync time we show average and maximum; maximum is useful to illustrate that worst case occupancy is sometime severe, which can limit performance.

The lock metric averages presented in the table are useful, but they hide the fact that locking behavior is not homogeneous. To illustrate more closely the locking effects that occur, we present a histogram of these two metrics as well. The histograms, such as Figure 6.29 show acquire latency ("Non-Contended") and hand-off latency ("Contended"). The bar height shows the *count* of lock acquisitions with latency (in microseconds) less than or equal to that listed below the bar. Note that both axes are logarithmic since the lock counts vary by such a wide margin, and because bars far to the right (i.e., long latency events) may contribute significantly to the overall performance even though they have much lower counts. Each column of plots represents a particular application, problem size, and machine size. By stacking them vertically it allows a straightforward comparison of the bins in the three related simulations.

Table 6.10: Water lock results summary: Each group of rows reflects one problem and machine size combination across the three different lock primitives. The remaining columns are as follows: compute time of the application; the percentage of the application compute time spent in synchronization (average and maximum across the processors); the average latency of a cache read miss; the fraction of time the protocol processor is busy (average and maximum as before); average acquire and hand-off latency as defined previously, with the fraction of locks encountering each case.

Lock Primitive	Compute Time	% of Time in Sync		Avg Read Miss	PP Occupancy		Acquire Latency (and fraction)	Hand-Off Latency (and fraction)
		Avg	Max		Avg	Max		
512 molecules, 16 processors, 19k lock acquisitions.								
LL/SC	304 ms	8%	18%	1.3 μs	2%	2%	1.2 μs *99%*	5.1 μs *1%*
MCS	304 ms	8%	17%	1.3 μs	2%	2%	1.7 μs *98%*	1.5 μs *2%*
FLASH	307 ms	8%	18%	1.1 μs	2%	2%	3.4 μs *98%*	1.4 μs *2%*
512 molecules, 32 processors, 35k lock acquisitions.								
LL/SC	165 ms	12%	22%	1.8 μs	4%	7%	1.8 μs *98%*	7.4 μs *2%*
MCS	163 ms	11%	21%	1.6 μs	3%	5%	1.9 μs *97%*	1.7 μs *3%*
FLASH	165 ms	11%	21%	1.2 μs	4%	5%	3.7 μs *93%*	1.7 μs *7%*
512 molecules, 64 processors, 69k lock acquisitions.								
LL/SC	110 ms	29%	37%	2.7 μs	6%	21%	2.9 μs *94%*	18.9 μs *6%*
MCS	100 ms	23%	32%	2.1 μs	6%	13%	2.2 μs *91%*	2.0 μs *9%*
FLASH	97 ms	18%	28%	1.5 μs	7%	13%	4.3 μs *84%*	2.0 μs *16%*
512 molecules, 128 processors, 135k lock acquisitions.								
LL/SC	148 ms	74%	79%	5.9 μs	5%	50%	8.3 μs *89%*	66.8 μs *11%*
MCS	78 ms	46%	54%	2.7 μs	7%	27%	2.7 μs *84%*	2.2 μs *16%*
FLASH	68 ms	37%	46%	1.8 μs	9%	18%	4.6 μs *75%*	2.3 μs *25%*
1024 molecules, 32 processors, 70k lock acquisitions.								
LL/SC	442 ms	11%	15%	1.4 μs	3%	6%	2.1 μs *98%*	13.8 μs *2%*
MCS	435 ms	10%	14%	1.4 μs	3%	10%	2.0 μs *97%*	1.4 μs *3%*
FLASH	440 ms	11%	15%	1.2 μs	2%	3%	3.8 μs *95%*	1.7 μs *5%*
1024 molecules, 64 processors, 136k lock acquisitions								
LL/SC	254 ms	19%	25%	2.1 μs	4%	10%	2.7 μs *96%*	21.4 μs *4%*
MCS	246 ms	15%	21%	2.0 μs	5%	25%	2.3 μs *93%*	2.0 μs *7%*
FLASH	246 ms	14%	20%	1.6 μs	6%	37%	4.4 μs *84%*	1.9 μs *16%*
1024 molecules, 128 processors, 273k lock acquisitions								
LL/SC	251 ms	44%	55%	7.2 μs	6%	45%	4.7 μs *89%*	42.2 μs *11%*
MCS	246 ms	40%	56%	8.0 μs	6%	51%	4.4 μs *79%*	4.1 μs *21%*
FLASH	189 ms	26%	39%	5.2 μs	7%	63%	4.9 μs *74%*	2.1 μs *26%*

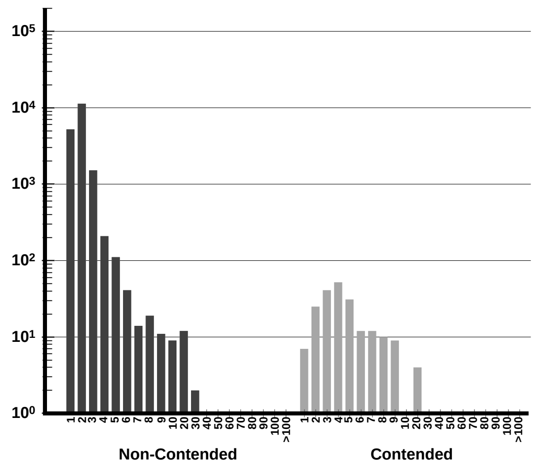


Figure 6.29: Water, 16 processors, 512 mols., LL/SC lock latency histograms.

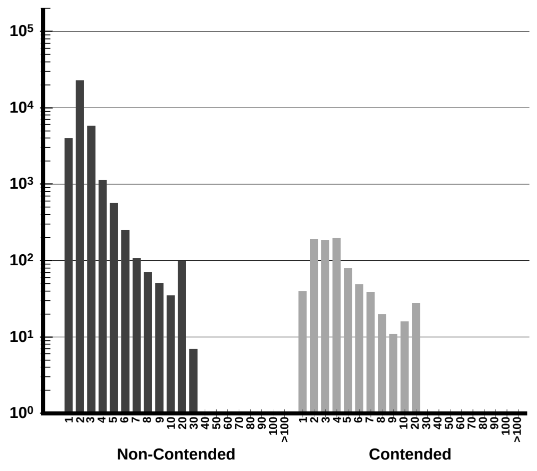


Figure 6.30: Water, 32 processors, 512 mols., LL/SC lock latency histograms.

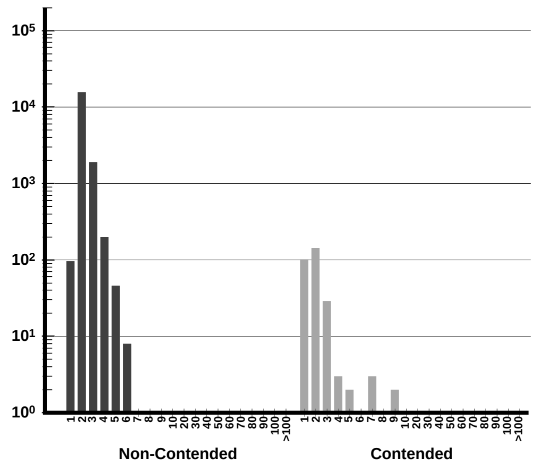


Figure 6.31: Water, 16 processors, 512 mols., MCS lock latency histograms.

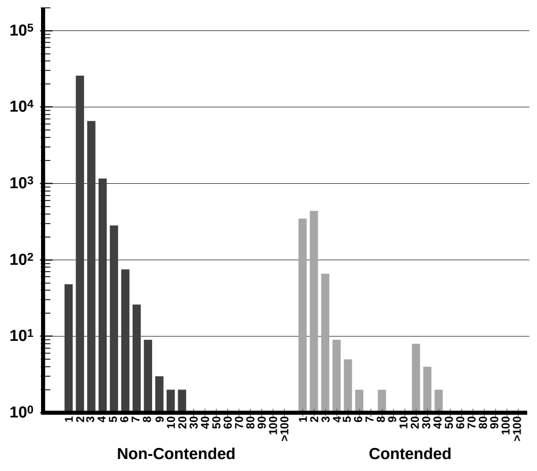


Figure 6.32: Water, 32 processors, 512 mols., MCS lock latency histograms.

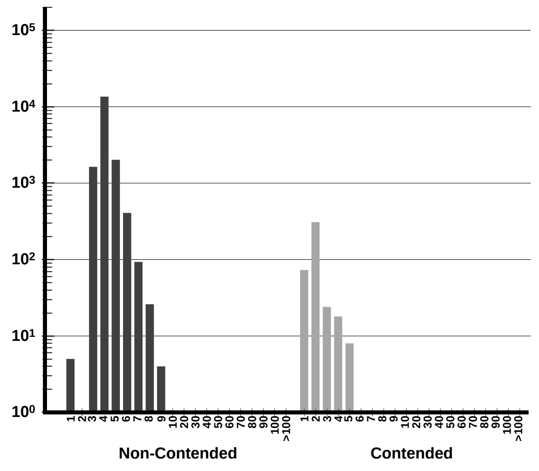


Figure 6.33: Water, 16 processors, 512 mols., FLASH lock latency histograms.

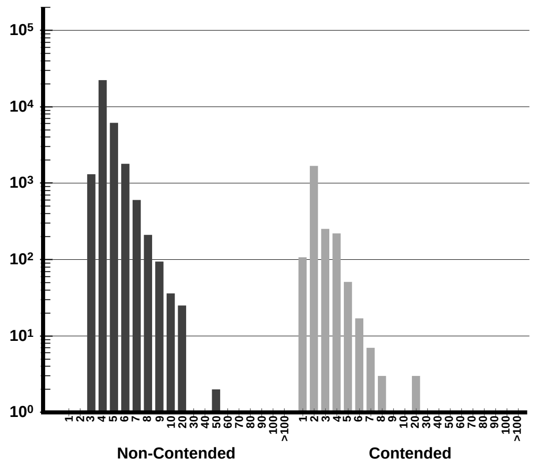


Figure 6.34: Water, 32 processors, 512 mols., FLASH lock latency histograms.

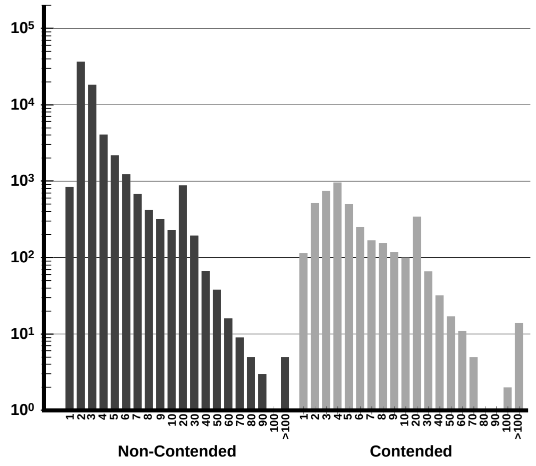


Figure 6.35: Water, 64 processors, 512 mols., LL/SC lock latency histograms.

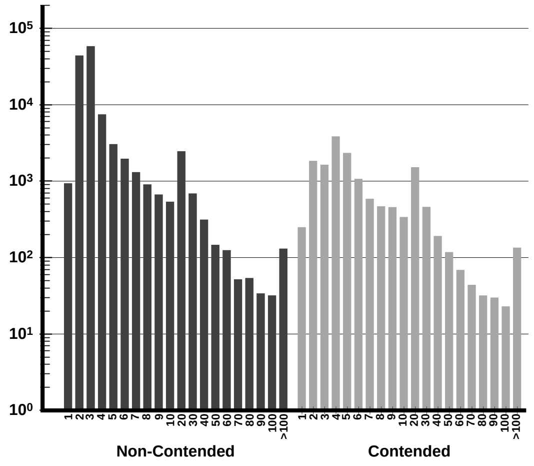


Figure 6.36: Water, 128 procs., 512 mols., LL/SC lock latency histograms.

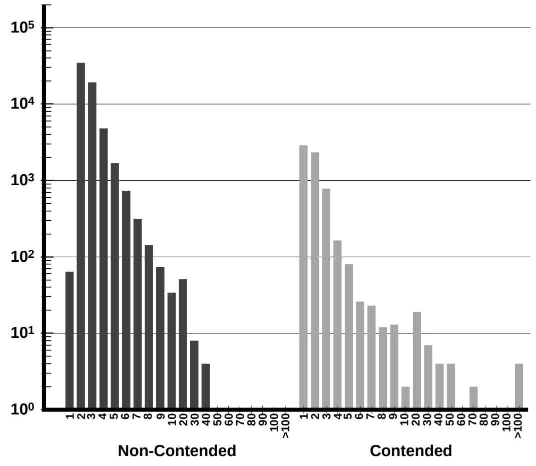


Figure 6.37: Water, 64 processors, 512 mols., MCS lock latency histograms.

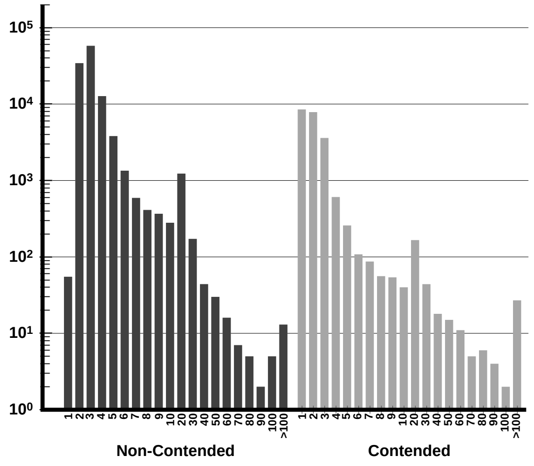


Figure 6.38: Water, 128 procs., 512 mols., MCS lock latency histograms.

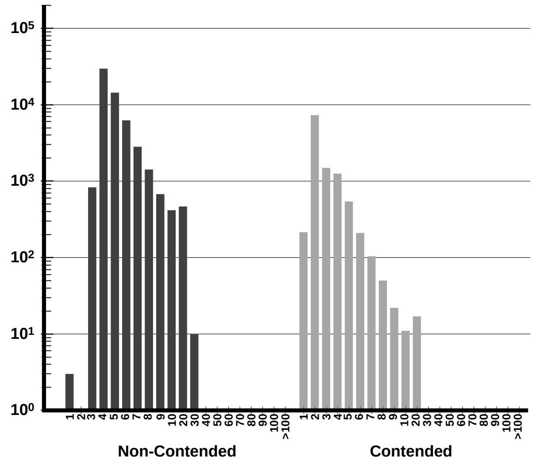


Figure 6.39: Water, 64 processors, 512 mols., FLASH lock latency histograms.

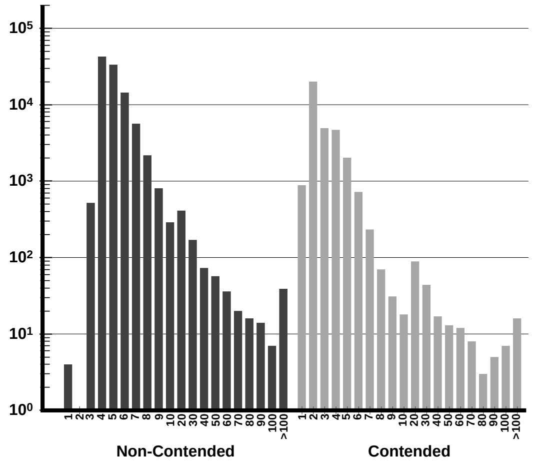


Figure 6.40: Water, 128 procs., 512 mols., FLASH lock latency histograms.

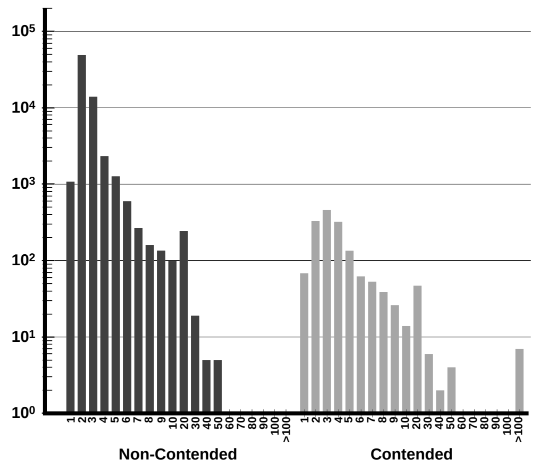


Figure 6.41: Water, 32 procs., 1024 mols., LL/SC lock latency histograms.

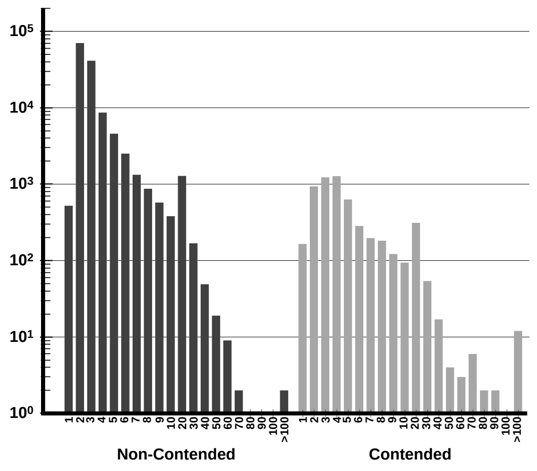


Figure 6.42: Water, 64 procs., 1024 mols., LL/SC lock latency histograms.

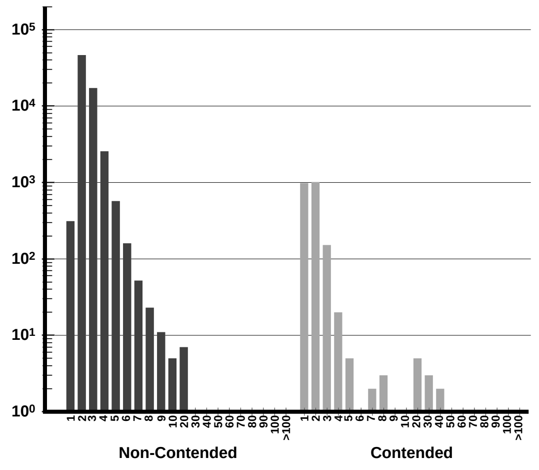


Figure 6.43: Water, 32 procs., 1024 mols., MCS lock latency histograms.

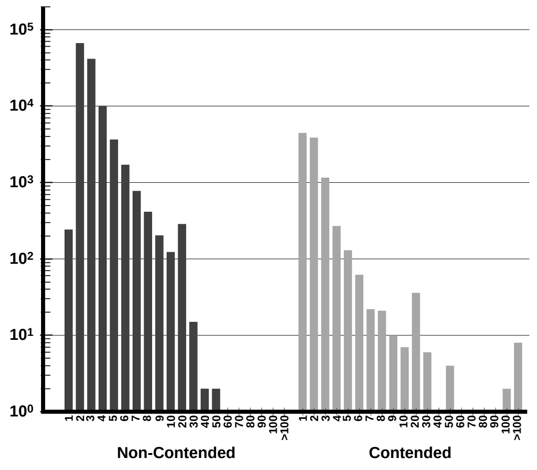


Figure 6.44: Water, 64 procs., 1024 mols., MCS lock latency histograms.

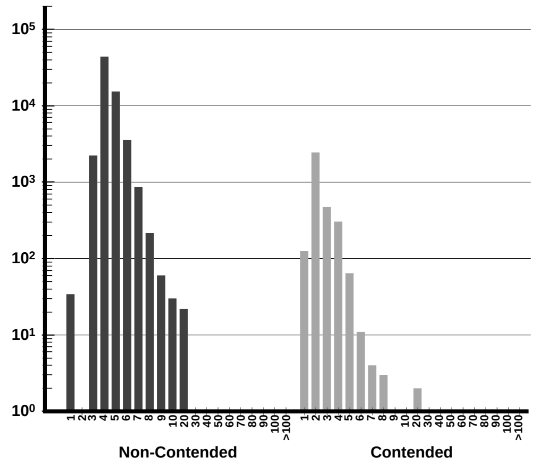


Figure 6.45: Water, 32 procs., 1024 mols., FLASH lock latency histograms.

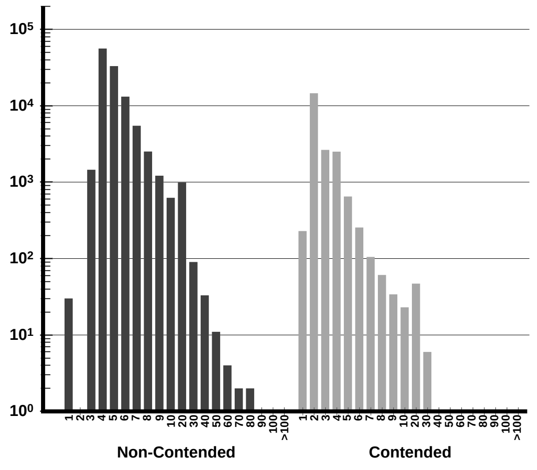


Figure 6.46: Water, 64 procs., 1024 mols., FLASH lock latency histograms.

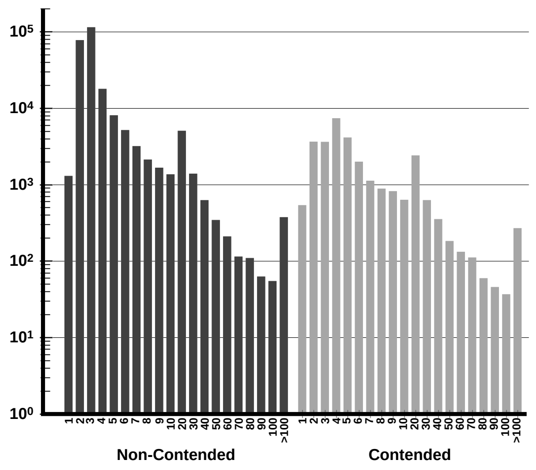


Figure 6.47: Water, 128 procs., 1024 mols., LL/SC lock latency histograms.

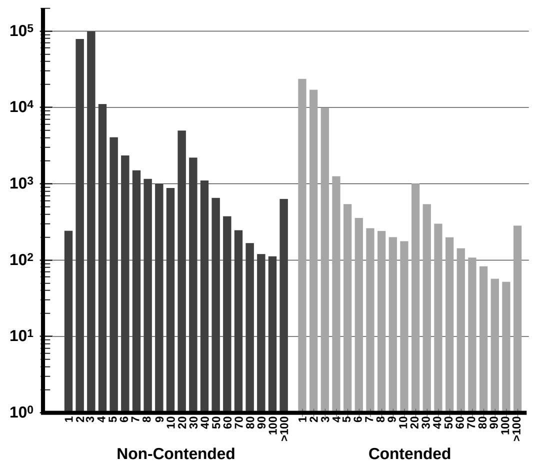


Figure 6.48: Water, 128 procs., 1024 mols., MCS lock latency histograms.

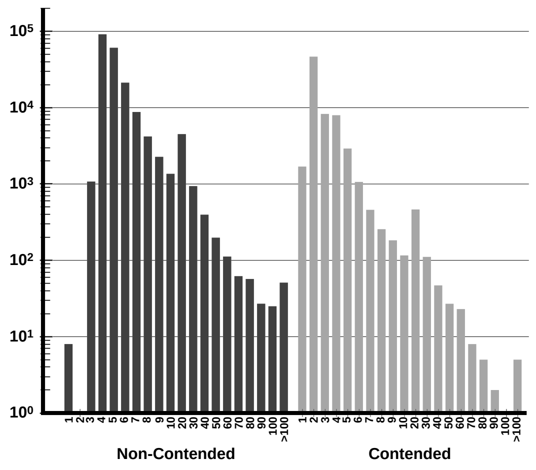


Figure 6.49: Water, 128 procs., 1024 mols., FLASH lock latency histograms.

Performance Analysis

We discuss the major effects in Water using the statistics table and histograms introduced above, beginning with small machine sizes and then incrementally considering the effects which arise as the system scales. For 16 and 32 processor machines in either problem size, lock performance is not a bottleneck for a number of reasons. The statistics table shows the total number of locks is fairly small. Each processor executes an essentially fixed number of locks per molecule, so lock counts grow with machine size. Processors also have ample work, so synchronization time is only about 15–20%, and PP occupancy is low. In these cases, the lock acquire latency metric tracks closely the microbenchmark predictions. Hand-off latency performs better than the worst case we established earlier, since every processor is not contending for the lock as in our microbenchmark. The lock metric histograms (columns under Figures 6.29, 6.30, and 6.41) show that locking is well-behaved, with events of any significant frequency taking less than 10 μs.

At 64 processors in both problem sizes, the application continues to scale reasonably well, though synchronization time begins to be more significant. This suggests that the application is beginning to reach its scalability limits by exhausting the highly parallelizable work that was sufficient for smaller machine sizes. PP occupancy has also increased to between 13–20%, though this still provides reasonable response time and thus average read miss latency increases only mildly. Both MCS and FLASH locks show mild latency gains over LL/SC at this size, as well as lower occupancy. We see that contended lock acquisitions remain efficient through queueing at about 2 μs, while LL/SC has degraded to 19 μs on average. The frequency of contended accesses for LL/SC locks is still low at this problem size, so the impact of this degradation is low. The lock metric histograms (columns under Figures 6.35, and 6.42) show the lock metrics more clearly. LL/SC encounters some long latencies for non-contended locks, but mostly achieves low latency and a fair number of cached re-acquisitions (the leftmost bar). MCS (Figure 6.37) improves the non-contended performance by reducing overall contention that causes long acquire latency. FLASH locks (Figure 6.39), also reduce very slow acquires, however, as our microbenchmarks indicated, the minimum acquire latency is over 2 μs (except for cached re-acquires).

Once the size reaches 128 processors, synchronization contention increases, causing performance to degrade for LL/SC, which is less efficient in that regime. For 512 molecules, LL/SC spends 74% of its time in synchronization, in part due to exponential back-off during contention. The non-synchronization portion of compute time is also longer than the other primitives, with high PP occupancy (50% worst case) from synchronization increasing average miss time considerably. The hand-off latency average of 67 μs is readily apparent from Figure 6.36, which indicates many locks take in excess of 100 μs to transition. Queued locking dramatically improves this situation: the hand-off latency average is essentially unchanged from 64 processors, and PP occupancy remains lower, improving average read miss latency. Figures 6.38 and 6.40 show that queued techniques encounter a sharp peak centered at 2 μs and only relatively few locks experience long waits (due to queueing for the PP). At this size, FLASH locks improve computation time by 12% over MCS, in part because more locks take the contended case where FLASH locks communicate more efficiently.

Part of the shift from non-contended to contended acquisitions is due to the fairness in queued lock techniques reducing the cached re-acquires as compared to LL/SC. This effect is visible in all the Water histograms in the leftmost non-contended histogram bar. The LL/SC lock implementation lacks fairness, and so a node can release and re-acquire a lock quickly, even if other processors are actively requesting it. Since the node acquired ownership to unlock, re-acquisition can succeed in the cache until other sharers request the line again. MCS locks provide fairness, though delays from contention to join the queue may allow some unfair re-acquisitions.

For 1024 molecules at 128 processors, the effects are similar to 512 molecules but are more significant, leading to a 23% improvement in execution time with FLASH locks as compared to MCS. Lock acquisition frequency is about the same as 512 molecules since the execution time doubles and so does the lock count. However, PP occupancy is extremely high at this size, and the average read miss time degrades considerably, especially for the conventional synchronization techniques. Accounting for the occupancy increase is the higher contention for locks at this size from two molecules sharing a lock. Despite the higher contention, FLASH acquire latency degrades only slightly compared to smaller runs because the algorithm is more communication-efficient. In contrast, Figure 6.48 shows that the high occupancy causes an increase in very slow MCS hand-offs, resulting in an average hand-off latency of 4.1 μs, the first significant degradation in MCS we see. We reiterate that the gains shown here for the 1024 molecule problem size are larger than would be encountered in practice since molecules share locks artificially. In fact, lock contention would probably *decrease* for a given machine size in Water if problem size and total lock count scaled together.

6.3.3 Barnes

Our study of Barnes is similar to that of Water, using the same application variants. Like Water, Barnes uses an array of locks to protect the data structures that track the bodies it simulates. We encountered similar contention problems due to central allocation of the lock array in Barnes, so we distributed this array as before, matching the distribution FLASH locks provide. The array of locks is limited to 512 locks as in Water.

Synchronization Usage

Barnes computes interaction forces in three dimensions between particles (or “bodies”), such as planets in a galaxy. Unlike the $O(n^2)$ algorithm used in Water to compute intermolecular forces, Barnes uses a hierarchical method which aggregates nearby groups of particles into their weighted

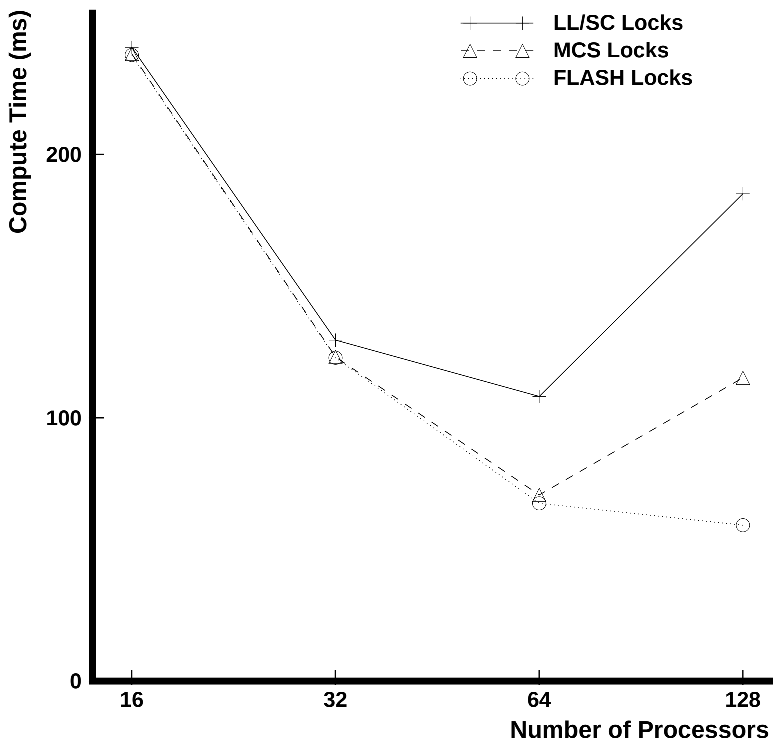


Figure 6.50: Barnes computation time using different lock techniques.

average. In this way, distant particles can reduce the computational complexity of calculating interactions while bounding the error that is introduced by adapting the level where aggregation is permitted. One symptom of the difference in this algorithm is that lock count in Barnes increases only about 25% from 16–128 processors, unlike Water where the count scales exponentially.

As introduced earlier, Barnes uses its array of locks for two functions: to protect updates to a group of particles and to protect a processor's data structures during the load balance phase where processor assignment is modified. Our simulations indicate that neither of these two phases is uniquely prone to contention; each contains some locks which experience high contention, but it is very non-uniform. This is probably due to the nature of the algorithm in which the communication patterns vary somewhat based on the particular traits of the bodies being simulated. Overall, Barnes lock acquires encounter contention somewhat more frequently than in Water, and FLASH locks perform well over the entire range of processors as a result.

Performance Analysis

We simulate Barnes running the 8192-body problem on machines from 16–128 processors. Our performance metric as before is computation time, but since initialization and cold-start time is significant we report the execution time of one phase of the application after it has reached steady state. The execution time is illustrated in Figure 6.50. Table 6.11 shows the statistics summary

Table 6.11: Barnes lock results summary. The columns use the same format explained in Table 6.10.

Lock Primitive	Compute Time	% of Time in Sync Avg	% of Time in Sync Max	Avg Read Miss	PP Occupancy Avg	PP Occupancy Max	Acquire Latency (and fraction)	Hand-Off Latency (and fraction)
8k bodies, 16 processors, 8.8k lock acquisitions.								
LL/SC	241 ms	3%	5%	1.0 μs	2%	4%	1.7 μs *95%*	6.2 μs *5%*
MCS	238 ms	2%	3%	0.9 μs	2%	3%	0.3 μs *93%*	1.3 μs *7%*
FLASH	237 ms	2%	3%	0.9 μs	2%	8%	0.5 μs *90%*	1.6 μs *10%*
8k bodies, 32 processors, 9.0k lock acquisitions.								
LL/SC	130 ms	9%	11%	1.3 μs	2%	7%	2.0 μs *93%*	14.2 μs *7%*
MCS	123 ms	5%	6%	1.1 μs	2%	5%	0.3 μs *89%*	1.8 μs *11%*
FLASH	123 ms	4%	6%	1.1 μs	3%	13%	0.6 μs *84%*	1.7 μs *16%*
8k bodies, 64 processors, 9.3k lock acquisitions								
LL/SC	108 ms	34%	42%	5.8 μs	4%	61%	2.1 μs *93%*	27.5 μs *7%*
MCS	71 ms	13%	16%	1.7 μs	5%	33%	0.4 μs *80%*	2.5 μs *20%*
FLASH	68 ms	9%	12%	1.4 μs	5%	28%	0.8 μs *78%*	1.6 μs *22%*
8k bodies, 128 processors, 10.0k lock acquisitions								
LL/SC	185 ms	49%	79%	22.9 μs	4%	63%	2.8 μs *88%*	42.3 μs *12%*
MCS	115 ms	39%	64%	14.7 μs	5%	61%	0.8 μs *53%*	3.2 μs *47%*
FLASH	59 ms	27%	42%	6.6 μs	6%	49%	1.8 μs *46%*	1.5μs *54%*

in the same format as Water. In Barnes, the FLASH lock version of the application achieves the best execution time at every problem size, though the gains only become significant beyond 64 processors.

We note first the conspicuous difference between Figures 6.53 and 6.55 for the 2 μs non-contended histogram bar. This effect, also present but less visible in the Water histograms, arises because the minimum FLASH acquire latency for a lock not already present is more than 2 μs, while MCS acquire latency is sometimes below 2 μs. The 1 μs bar for both cases represent rapid cached re-acquisition of a previously held lock, which occur very frequently in Barnes. Table 6.11 lists the average of these two effects for acquire latency, which is why it appears to be less than the microbenchmark predictions.

Beginning with the 16 processor simulation, Figure 6.51 shows that even at this small size, LL/SC locks perform poorly and encounter a wide range of lock latencies. The queued lock primitives achieve much more uniform behavior, and perform similarly. At this size the lock metrics favor MCS slightly, though the average miss latency is slightly lower for FLASH (beyond the precision shown in the table), accounting for the marginally faster execution time. Similar effects are

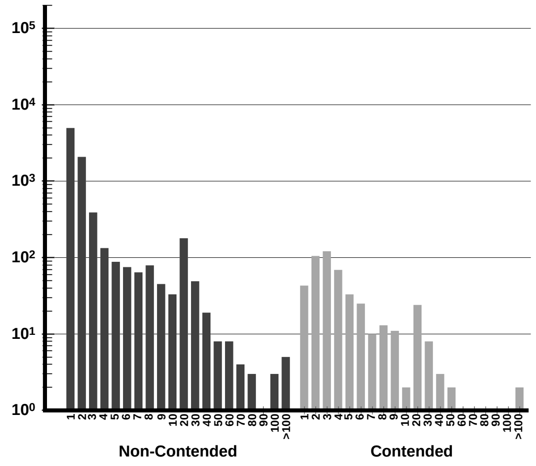


Figure 6.51: Barnes, 16 procs., 8k bodies, LL/SC lock latency histograms.

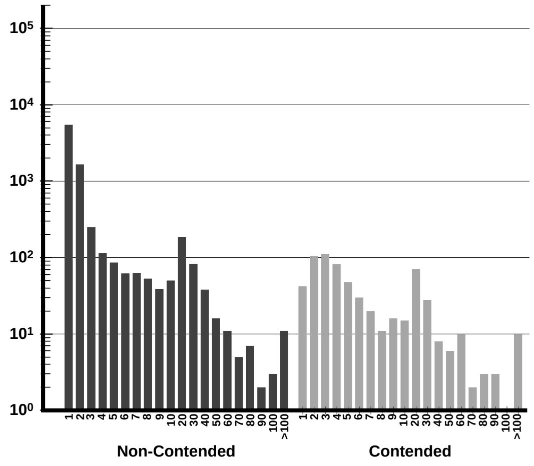


Figure 6.52: Barnes, 32 procs., 8k bodies, LL/SC lock latency histograms.

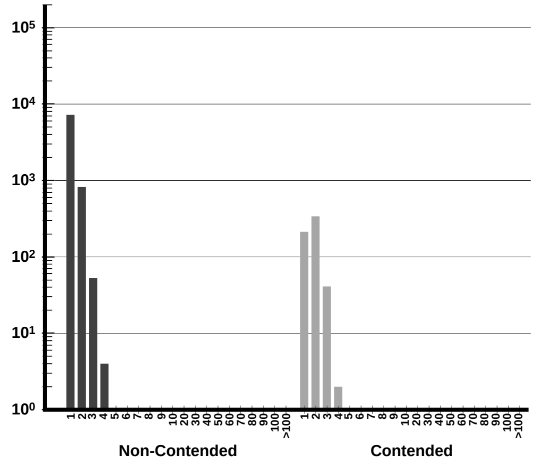


Figure 6.53: Barnes, 16 procs., 8k bodies, MCS lock latency histograms.

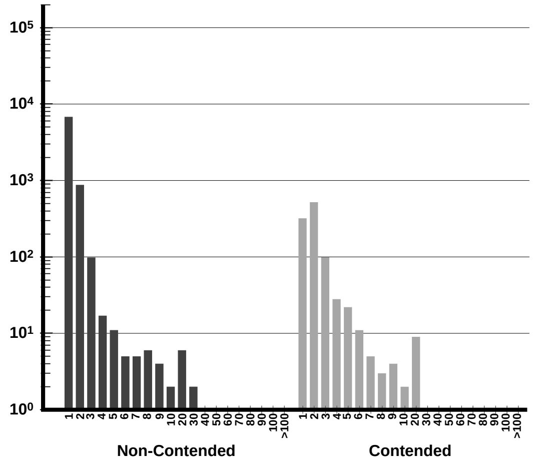


Figure 6.54: Barnes, 32 procs., 8k bodies, MCS lock latency histograms.

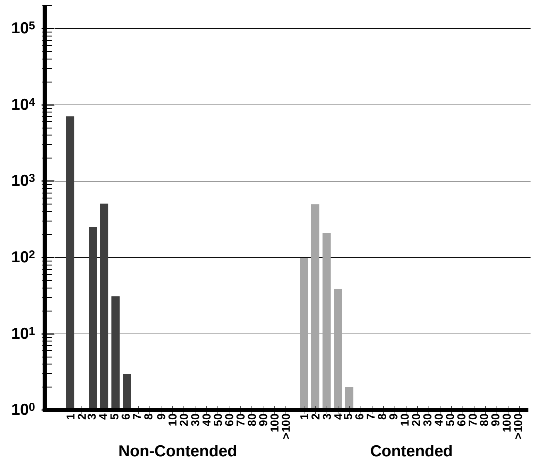


Figure 6.55: Barnes, 16 procs., 8k bodies, FLASH lock latency histograms.

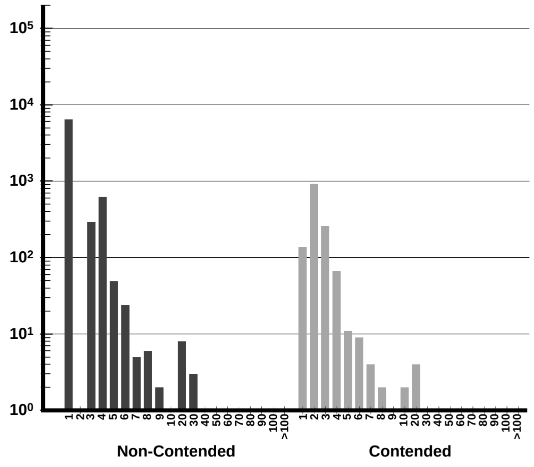


Figure 6.56: Barnes, 32 procs., 8k bodies, FLASH lock latency histograms.

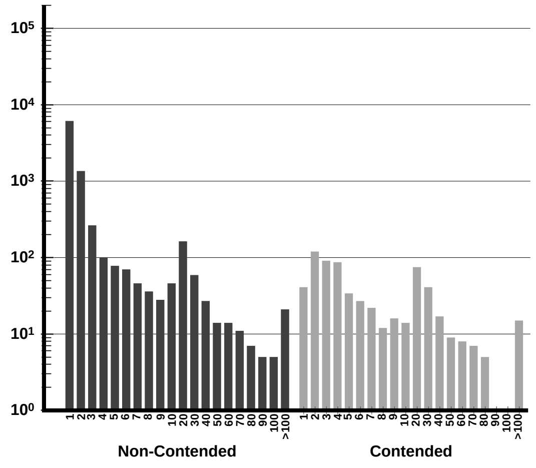


Figure 6.57: Barnes, 64 procs., 8k bodies, LL/SC lock latency histograms.

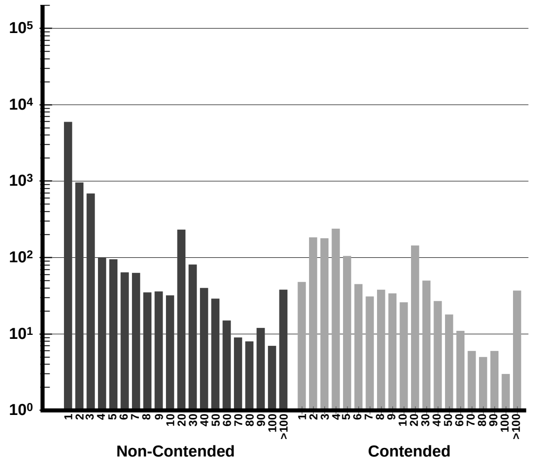


Figure 6.58: Barnes, 128 procs., 8k bodies, LL/SC lock latency histograms.

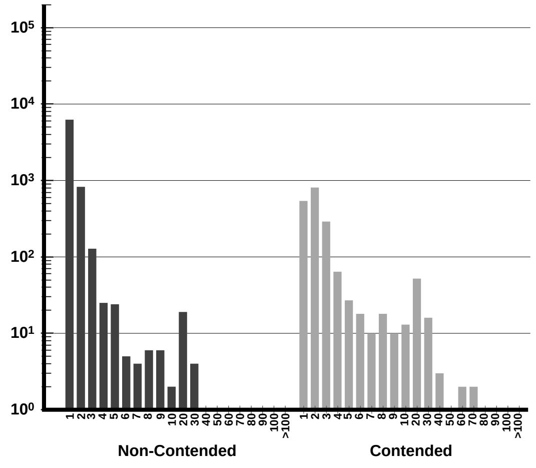


Figure 6.59: Barnes, 64 procs., 8k bodies, MCS lock latency histograms.

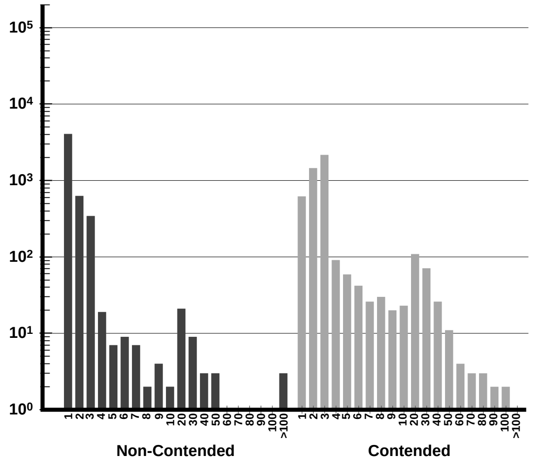


Figure 6.60: Barnes, 128 procs., 8k bodies, MCS lock latency histograms.

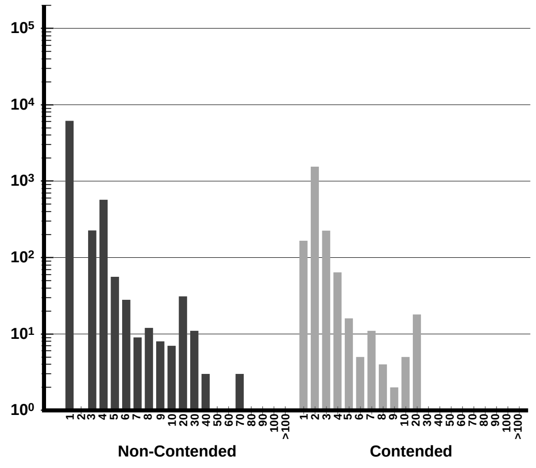


Figure 6.61: Barnes, 64 procs., 8k bodies, FLASH lock latency histograms.

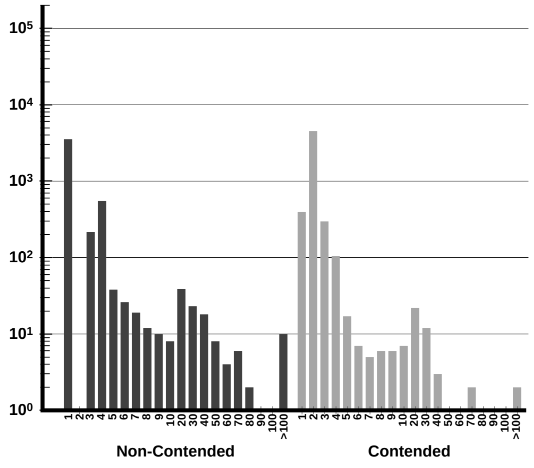


Figure 6.62: Barnes, 128 procs., 8k bodies, FLASH lock latency histograms.

visible at 32 processors; the higher PP occupancy for FLASH at that size may be due to less efficient non-contended lock handling, though contended performance begins to surpass MCS locks.

At 64 processors, FLASH contended lock performance actually improves as compared to 32 processors, while MCS locks and LL/SC continue to degrade. PP occupancy for the shared memory lock approaches increases as well, particularly LL/SC, coupled with an increase in cache miss latency. These effects combine to yield a 4% execution time advantage for FLASH locks. In all primitives, non-contended performance degrades slightly, though cached re-acquisitions remain a large component and thus the average latency remains low.

For the 128 processor system, the trend of these effects continues, causing the execution time both shared memory lock primitives to slow as compared to 64 processors. In contrast, the version using FLASH locks continues to scale, achieving a 49% gain versus MCS locks at 128 processors and a 16% gain versus the best case achieved by shared memory locks (MCS locks at 64 processors). A major reason for this result is the shift in lock acquisitions to the contended case, which occurs for each scaling step but is particularly prevalent at 128 processors. At this size, LL/SC contended performance is nearly a factor of 30 slower than FLASH locks, and MCS is twice as slow, so the increase in contended lock acquisitions enables appreciable gains from FLASH. Contended FLASH locks also eliminate artifactual communication that reduces protocol processor congestion and improves average cache miss latency. The 128 processor simulations show significant queueing effects in the outgoing network queues (not shown in the table) and an increase in negative acknowledgements as a result. Reducing PP occupancy helps abate both effects and improves performance noticeably.

Note that at this large machine size, the results from Barnes are somewhat unpredictable since the system is spending a significant amount of time in synchronization. Though FLASH locks show clear gains at this size, we observe that Barnes might employ different synchronization primitives or restructure the algorithms somewhat if execution at this operating point were desired.

6.3.4 Ocean

For the Ocean application, we focus our study on barriers. The baseline version is based on the tournament tree barrier we describe earlier, since the elementary LL/SC barrier performs so poorly it is unusable. We generate two other versions, one using the MCS barrier, the other using the FLASH barrier presented in Section 6.2.

For each barrier primitive we use the configuration that our microbenchmark results show performs approximately the best, using as a metric $t_{\text{ft}} + t_{\text{rel}}$ since that represents the worst case release from the barrier. For Tournament barriers we use radix 2 trees, and for MCS barriers we use radix 4. The FLASH barrier uses a 4-ary tree even though 5-ary is slightly faster in an attempt to distribute the tree more evenly over the processors and reduce the likelihood that the outgoing network queue fills, requiring recovery using the software queue.

Synchronization Usage

Ocean is an iterative algorithm with ten separate phases per iteration. Barriers are used between every phase to assure results are completed before they are read by other processors. They are also used within the multigrid solver to separate its internal phases, in fact the solver's barriers comprise about 85% of the barriers in the application. Overall, the application uses about 150 barriers in the region we study.

Ocean encounters a wide range of phase lengths, ranging from very short ones that update a global sum to longer ones that do significant communication and computation. In the multigrid solver phases, immediately after the barrier it copies either the red or black portion of the grid, then executes the relaxation phase on the other. The subsequent phase then does the same on the opposite portions of the grid. These grid copy phases utilize extensive prefetching that causes intense communication immediately after the barrier. This characteristic has a significant impact on the performance effects we see.

Its frequent barriers, combined with the presence of some very short phases offers the potential for some performance improvement from optimizing barriers. Our results match this expectation in some cases, especially larger machine sizes. However, the characteristics of the application's algorithms, especially the multigrid solver, makes Ocean very sensitive to communication performance within its phases. Though improving barrier performance is one important aspect, optimized barriers may also increase contention if different processors interact differently or operate more closely in lock step as a result. Due to these balancing effects we find the execution time gains from barriers vary by problem size, even though the barrier metrics themselves measured from the application run show unequivocal gains from FLASH barriers.

Simulation Results Format

We run Ocean at two problem sizes: 258x258 over 16–64 processors and 514x514 for 32–64 processors. As our primary metric for Ocean, we report execution time for five phases of the application, excluding the first phase so that the application is executing in steady state. If the application were executed for more time steps than simulation permits, this time should scale linearly. The execution time results are presented in Figure 6.63. Like the previous applications, we show a range of statistics in Table 6.12 with the leftmost seven columns characterizing the execution time, synchronization time, and occupancy characteristics as before. In this case the two rightmost columns show averages of the barrier metrics introduced in Section 6.2.1.

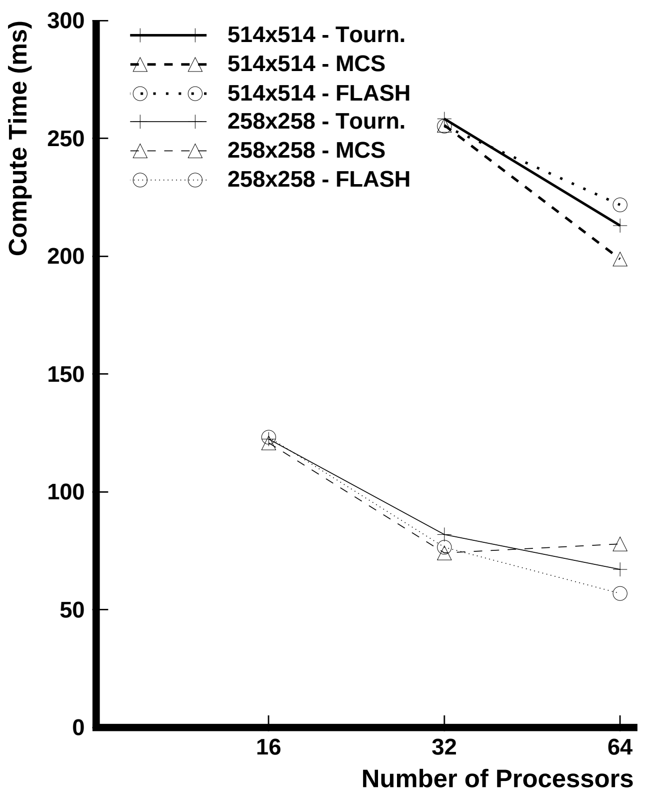


Figure 6.63: Ocean computation time using different barrier techniques.

Just as in locks, we find that the barrier performance varies somewhat during the run based on the different places it is used, so we also present a histogram of fall-though and release latency. The histograms use the same format as before, except that the y-axis is now linear due to the significantly smaller number of barriers than locks.

Unlike the microbenchmark analysis in which arrivals at the barrier were carefully controlled, in this case the arrival characteristics are determined largely by the application. We introduce two additional metrics in our analysis to analyze the arrivals since, as we showed earlier, the arrival characteristics can significantly affect barrier performance. The first is *last arrival interval*, which measures the duration between the last and next-to-last arrival (in the nomenclature of Section 6.2.1, this can be expressed as $t_{J_n} - t_{J_{n-1}}$). This metric provides some indication of the severity of arrival contention: when last arrival interval is large, the late arrival case approximately applies. However, when it is small, this metric does not show how many other arrivals were nearby. The second metric, called *all arrival interval*, addresses this question somewhat by measuring the duration between the first and last arrival (in the nomenclature of Section 6.2.1, $t_{J_n} - t_{J_1}$). It provides an indication of the arrival spread (to gauge the overall contention) and also shows the load imbalance between the processors. Comparing synchronization time between runs of the same size is also useful to gauge

Table 6.12: Ocean barrier lock results summary. The leftmost seven columns use the same format explained in Table 6.10. The remaining columns show barrier-specific results (all in microseconds): the average interval between the last and next-to-last barrier arrival; the average interval between the first and last barrier arrival; the median and mean for each of the fall-through and release latency metrics as defined earlier .

Barrier Primit.	Compute Time	% of Time in Sync Avg	% of Time in Sync Max	Avg. Read Miss	PP Occupancy Avg	PP Occupancy Max	Last Arriv. Interv. (μs)	All Arriv. Interv. (μs)	Fall-Through Latency Med. (μs)	Fall-Through Latency Mean (μs)	Release Latency Med. (μs)	Release Latency Mean (μs)
258x258 Ocean, 16 processors, 143 barriers.												
Tourn	122 ms	14%	19%	1.0 μs	14%	19%	17	165	11.2	10.7	19.4	21.3
MCS	121 ms	12%	16%	1.1 μs	14%	20%	26	157	10.8	11.5	9.2	11.2
FLASH	123 ms	13%	18%	1.1 μs	14%	19%	41	182	2.4	2.6	1.2	1.4
258x258 Ocean, 32 processors, 143 barriers.												
Tourn	82 ms	27%	33%	4.2 μs	12%	27%	28	211	12.3	11.9	31.1	41.4
MCS	74 ms	23%	33%	3.6 μs	12%	24%	11	172	14.0	13.8	14.9	23.4
FLASH	77 ms	20%	31%	5.1 μs	12%	25%	11	199	2.6	2.9	2.1	3.1
258x258 Ocean, 64 processors, 143 barriers.												
Tourn	67 ms	44%	57%	5.1 μs	14%	39%	19	263	15.6	15.4	40.0	50.7
MCS	78 ms	45%	58%	7.8 μs	12%	32%	20	332	14.5	15.4	23.1	29.4
FLASH	57 ms	41%	50%	4.3 μs	15%	33%	16	209	3.0	4.5	2.1	4.0
514x514 Ocean, 32 processors, 156 barriers.												
Tourn	258 ms	14%	18%	0.7 μs	15%	22%	30	326	13.9	13.8	28.8	36.3
MCS	256 ms	13%	16%	0.7 μs	15%	23%	38	311	13.7	14.0	14.6	20.5
FLASH	255 ms	12%	16%	0.7 μs	15%	23%	37	319	3.0	3.3	2.1	2.1
514x514 Ocean, 64 processors, 156 barriers.												
Tourn	213 ms	31%	42%	5.1 μs	12%	31%	56	640	16.0	15.9	41.7	59.0
MCS	199 ms	30%	39%	4.0 μs	12%	32%	71	598	14.2	15.2	26.2	62.7
FLASH	222 ms	31%	46%	5.4 μs	11%	40%	78	735	3.0	3.3	2.1	4.8

load imbalance, since it gives another indication of processors waiting at the barrier while others are still working.

Our results show that these auxiliary metrics have heterogeneous characteristics (even more so than fall-through and release latency), so we illustrate them with histograms as well. For the Ocean results, therefore, a single page contains two columns of plots for a particular problem size, barrier metrics on the left, and arrival characteristics on the right.

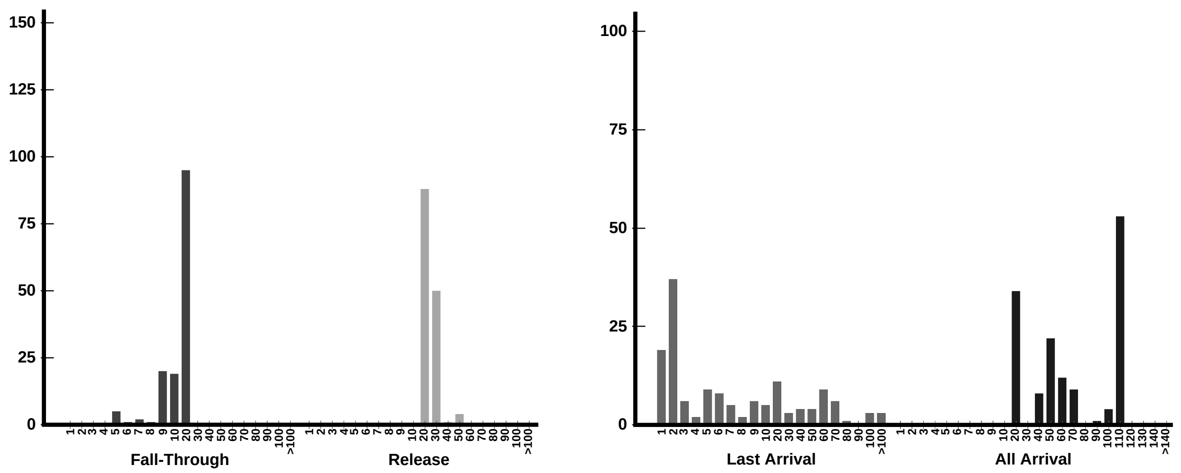


Figure 6.64: Ocean, 16 procs., 258x258, Tourn. barrier metric *(left)* and arrival *(right)* histograms.

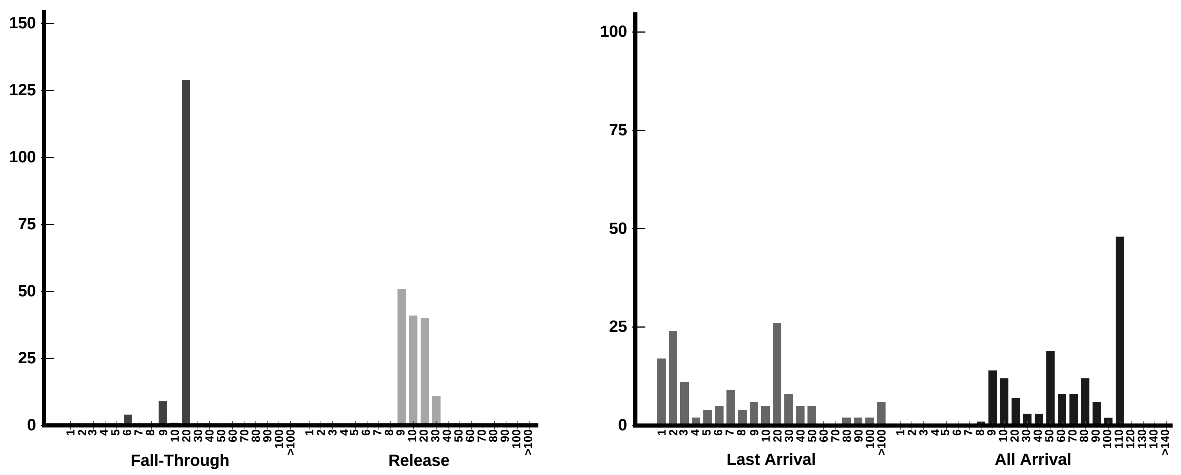


Figure 6.65: Ocean, 16 procs., 258x258, MCS barrier metric *(left)* and arrival *(right)* histograms.

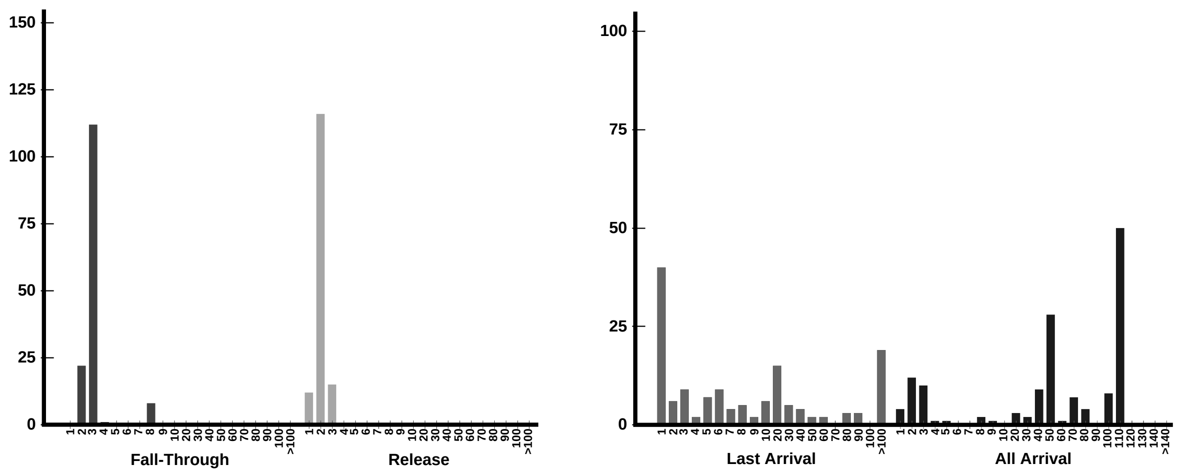


Figure 6.66: Ocean, 16 procs., 258x258, FLASH barrier metric *(left)* and arrival *(right)* histograms.

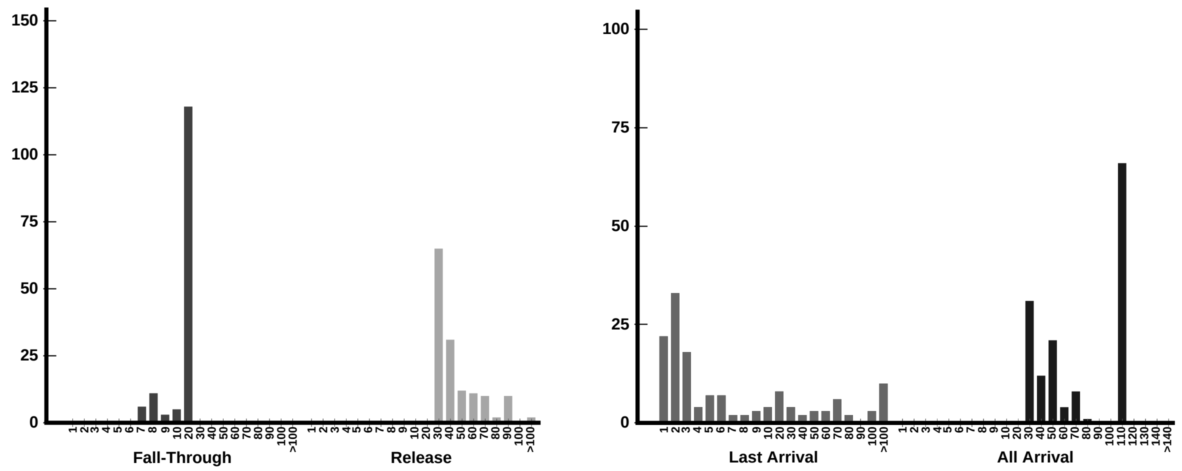


Figure 6.67: Ocean, 32 procs., 258x258, Tourn. barrier metric *(left)* and arrival *(right)* histograms.

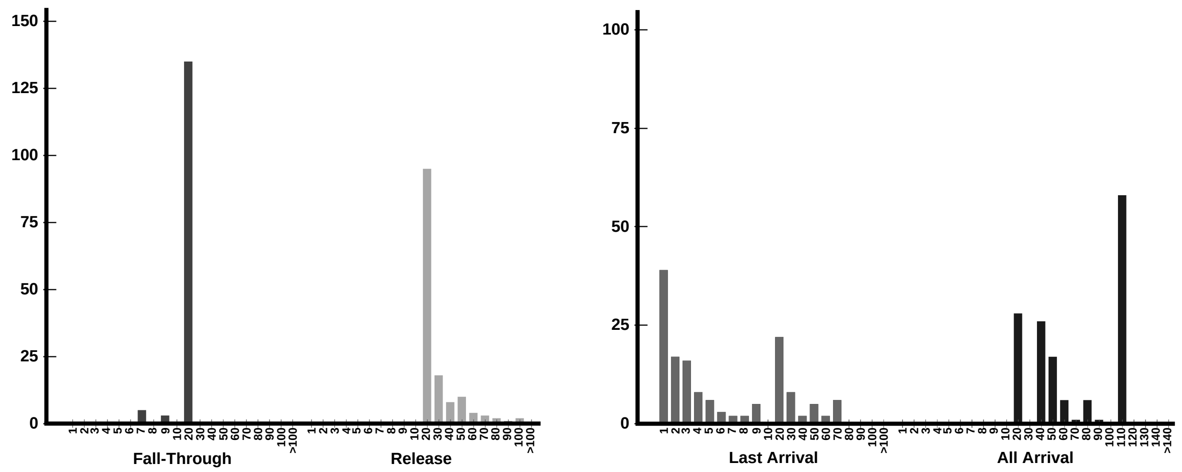


Figure 6.68: Ocean, 32 procs., 258x258, MCS barrier metric *(left)* and arrival *(right)* histograms.

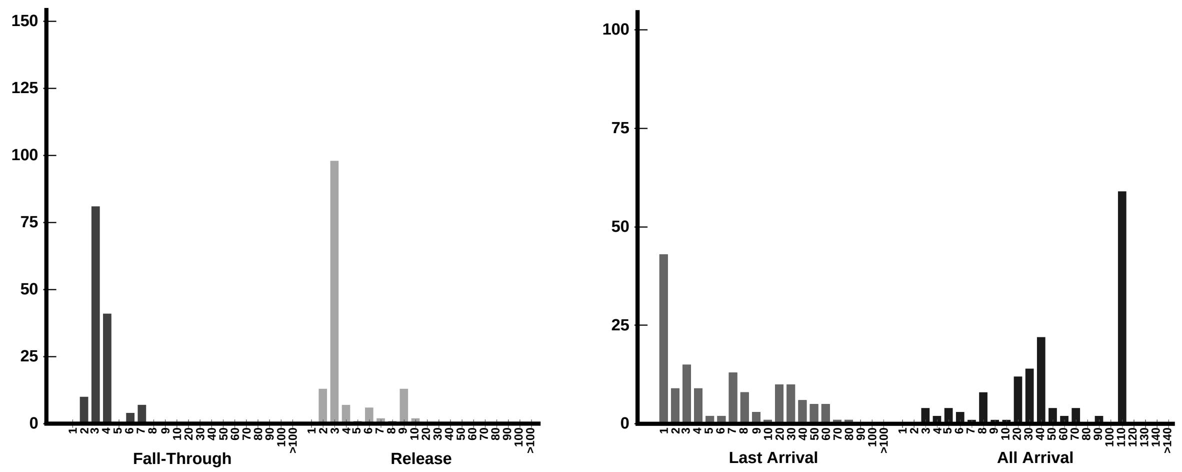
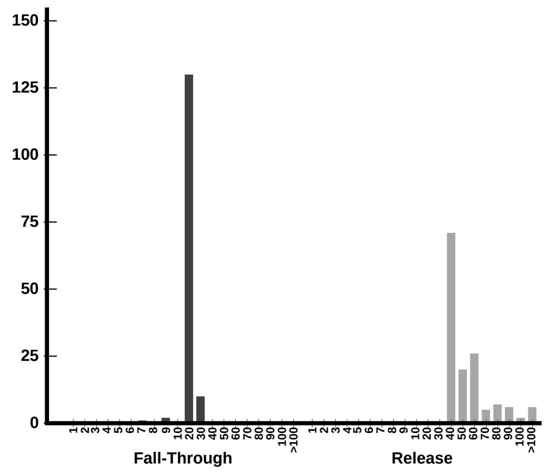


Figure 6.69: Ocean, 32 procs., 258x258, FLASH barrier metric *(left)* and arrival *(right)* histograms.



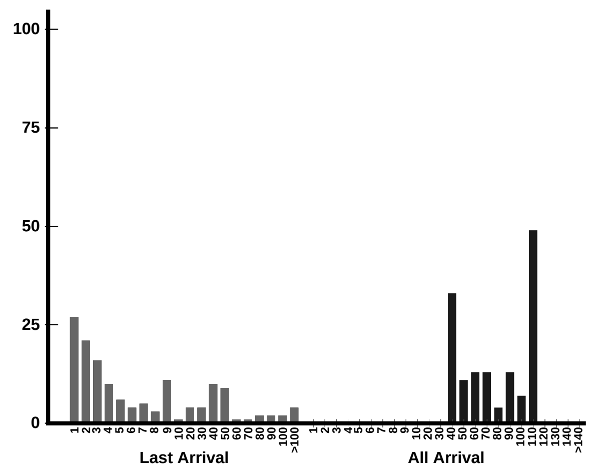
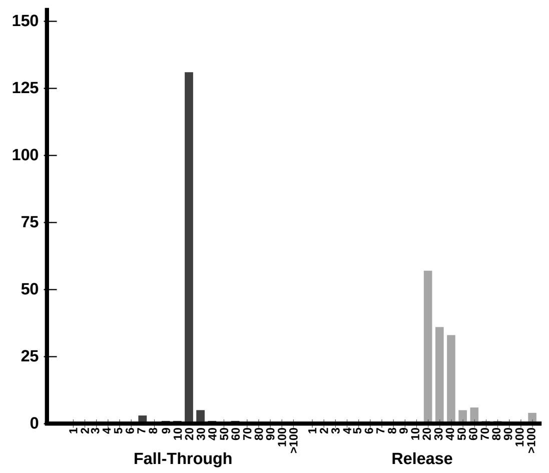


Figure 6.70: Ocean, 64 procs., 258x258, Tourn. barrier metric *(left)* and arrival *(right)* histograms.



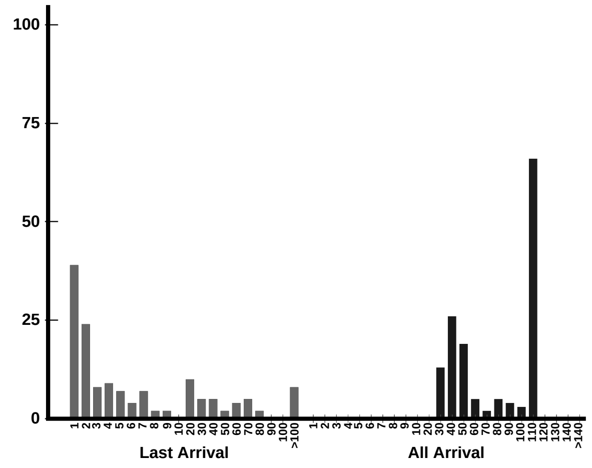
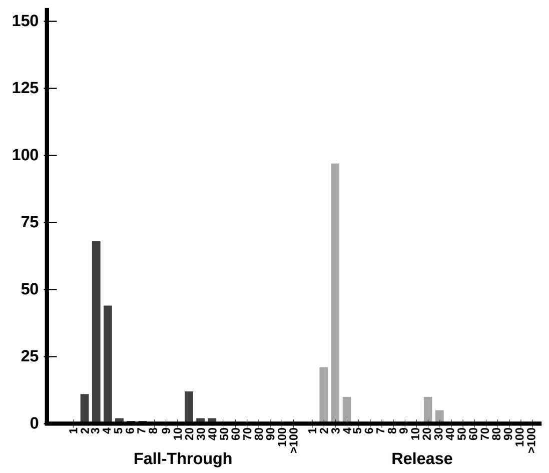


Figure 6.71: Ocean, 64 procs., 258x258, MCS barrier metric *(left)* and arrival *(right)* histograms.



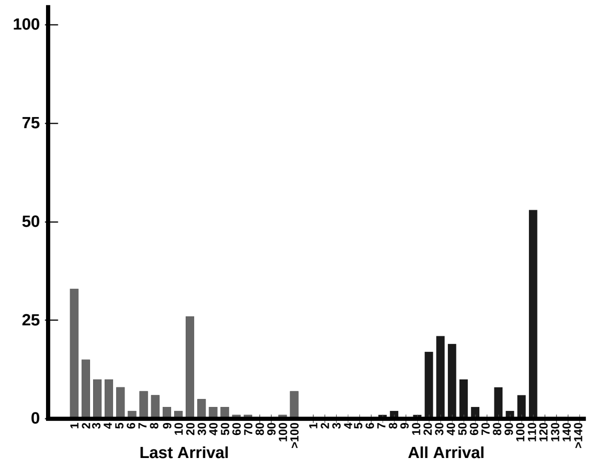


Figure 6.72: Ocean, 64 procs., 258x258, FLASH barrier metric *(left)* and arrival *(right)* histograms.

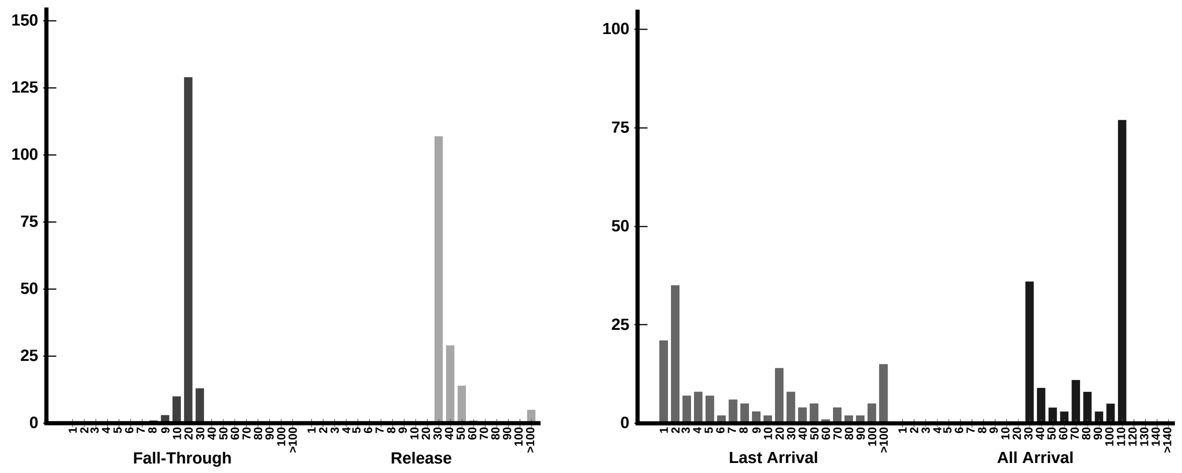


Figure 6.73: Ocean, 32 procs., 514x514, Tourn. barrier metric *(left)* and arrival *(right)* histograms.

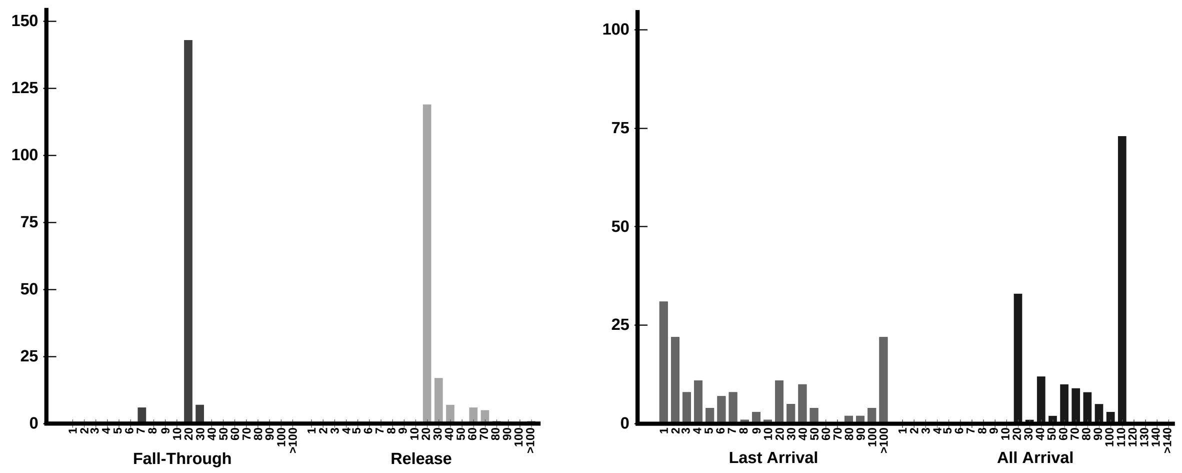


Figure 6.74: Ocean, 32 procs., 514x514, MCS barrier metric *(left)* and arrival *(right)* histograms.

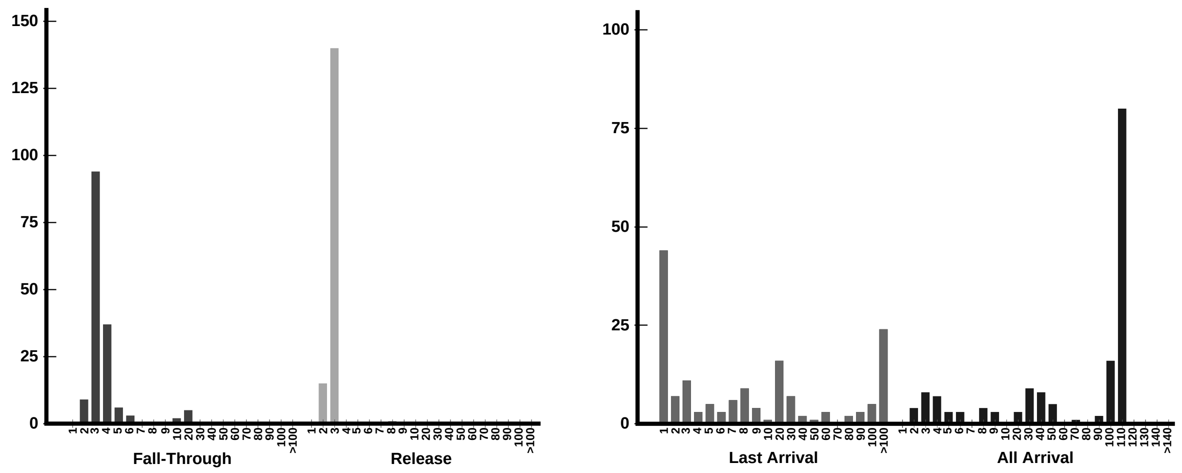
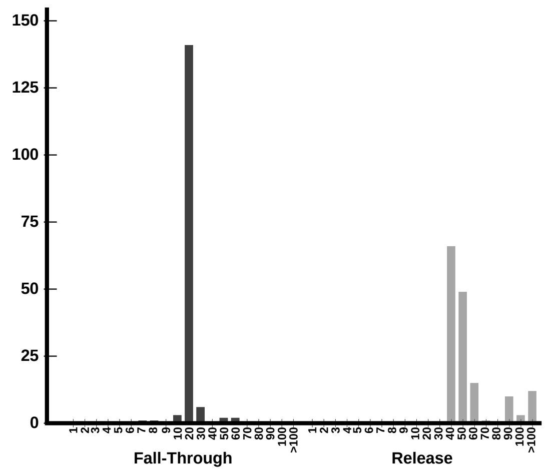


Figure 6.75: Ocean, 32 procs., 514x514, FLASH barrier metric *(left)* and arrival *(right)* histograms.



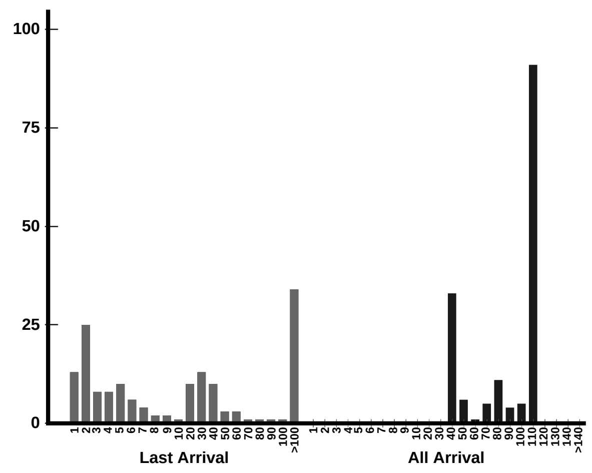
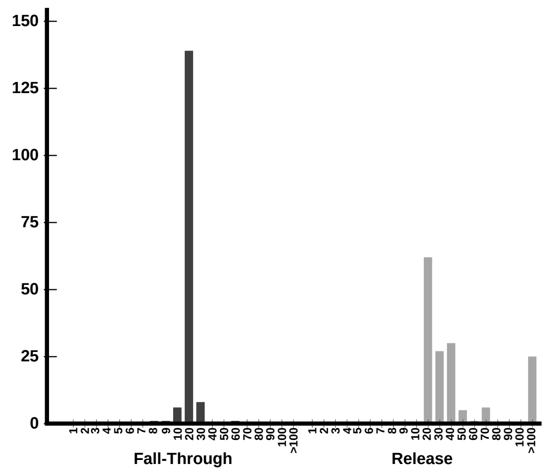


Figure 6.76: Ocean, 64 procs., 514x514, Tourn. barrier metric *(left)* and arrival *(right)* histograms.



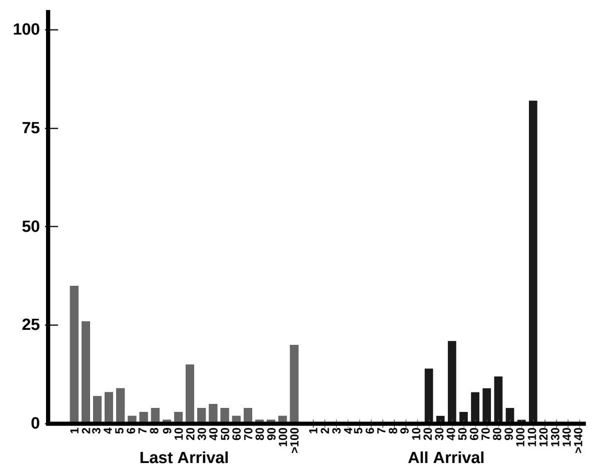
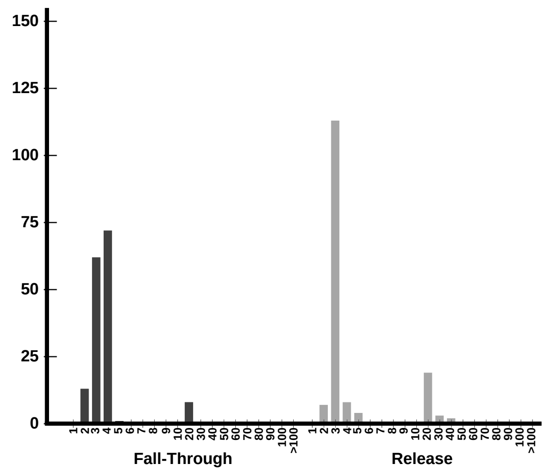


Figure 6.77: Ocean, 64 procs., 514x514, MCS barrier metric *(left)* and arrival *(right)* histograms.



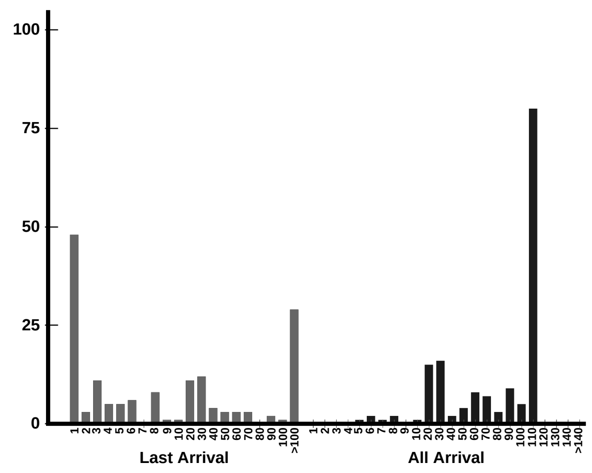


Figure 6.78: Ocean, 64 procs., 514x514, FLASH barrier metric *(left)* and arrival *(right)* histograms.

Barrier Performance Analysis

Unlike the previous applications we study, Ocean contains a sufficient number of barriers to allow us to study their performance adequately in the context of real application behavior. We begin by considering the performance of the barrier primitive itself, using the metrics t_{ft} and t_{rel} and comparing to the results of the microbenchmark analysis. Then we return to the larger issue of analyzing the performance impact of barriers on the application overall.

Table 6.12 shows that based on the performance metrics, FLASH barriers are superior to the other techniques in every case. As predicted, the gains are between a factor of 4 up to more than an order of magnitude. Beginning with fall-through latency, t_{ft}, to compare with the microbenchmarks we must first determine the nature of the barrier arrivals. For this we use the last arrival interval histograms (such as the column of figures under Figure 6.64, right). These indicate that about half the time the final barrier arrivals occur less than 3 μs of each other, corresponding to the simultaneous arrival case. The other half encounter about 3 μs or more between the final two arrivals, which corresponds roughly to the late arrival case. The results in Table 6.12 show that t_{ft} matches this analysis and is approximately the average of the late and simultaneous arrival predictions from the microbenchmarks. The barrier metric histograms (such as the column of figures under Figure 6.64, left) reflect this as well. Though the histogram granularity does not show this in detail, the pronounced fall-through latency peak reflects a fairly smooth distribution between the two extremes of late and simultaneous arrival when examined at higher resolution.

The release latency also corresponds fairly closely to the microbenchmark predictions. In most cases the latency increases slightly, due to traffic introduced into the system by the earliest released processors. The microbenchmarks isolated the barrier itself and thus did not encounter interference of this kind. The release latency histograms show the performance of the three primitives in more detail. The most notable effect is that the release performance of the software techniques is less resilient to contention degrading significantly in some cases. FLASH barriers achieve release performance close to the predictions, and encounter much less degradation from contention.

Returning to fall-through latency, we notice that a small fraction of the FLASH barriers have longer latency than the microbenchmark predictions. Figure 6.72 (left) shows this effect clearly in the 20 μs histogram bar (i.e., fall-through latency between 10–20 μs). This arises from the application and coherence protocol: in the relaxed consistency mode we use, invalidation acknowledgements for exclusive ownership requests can be collected in the background after a write has completed. Thus, the processor may complete its work and reach the barrier while invalidations are still outstanding. Internally, the FLASH barrier executes an uncached read to communicate with MAGIC. This read causes an implicit memory fence by the processor that forces all invalidations to be counted before the read can issue. For consistency, and to account for all the overheads of the different primitives we study, we count barrier fall-through latency from the start of the final processor's barrier routine, and thus the delay while the invalidation acknowledgements arrive is counted towards the fall-through latency. This effect is unavoidable given the coherence protocol, and reflects one part of the fundamental synchronization the barrier provides.

In Table 6.12 we show both mean and median barrier metrics to illustrate that the median case is in line with the predictions, while the mean is perturbed slightly by instances of this effect. Though the other barrier primitives are subject to this effect as well, we notice that in practice they do not encounter as many slow events as FLASH barriers. While this effect is deeply dependent on the ordering and timing of requests within the application, it is likely that the significantly more uniform release characteristics of FLASH barriers is responsible for arrivals at the subsequent barrier with more acknowledgements outstanding.

Overall Performance Analysis

Now that we have analyzed the barrier performance specifically, we consider the application as a whole to understand why its performance does not track the clear improvements from FLASH barriers. The most important observation is that even though the standard barrier primitives are far less efficient than FLASH barriers, they still account for only a fraction of the execution time. The impact of the primitive itself for n barriers is bounded between $n(t_{\text{ft}})$ and $n(t_{\text{ft}} + t_{\text{rel}})$, since every processor must wait t_{ft}, but only the worst case processor waits $t_{\text{ft}} + t_{\text{rel}}$. If we consider the tournament barrier as a reference point, across the range of simulations the lower bound is always less than 3%, and the upper bound is less than 10% in nearly every case. Only for the 64 processor/258 problem size does the maximum barrier contribution reach about 14%. Thus, the barrier primitive's performance itself is only responsible for some of the application-level performance differences in our results. We also encounter differences in the *application's* execution resulting from the synchronization primitives.

Looking first at the 258x258 problem size in Table 6.12, we see that the FLASH barrier translates to improved application performance overall, especially as machine size grows. Notice that all arrival interval, which provides an indication of load balance, deteriorates markedly with the other barriers, while with FLASH barriers it stays fairly constant. At 64 processors FLASH also reduces average read miss latency by reducing the communication artifacts due to barriers. These effects combine to provide an overall execution time improvement at 64 processors of 14% versus tournament barriers and 26% versus MCS.

For the 514x514 problem size, however, we see very different behavior. On the one hand, barrier performance metrics indicate FLASH barriers clearly outperform the other primitives for this problem size as well. Unfortunately, we also find increases in average read miss latency, PP occupancy, and all arrival latency for FLASH barriers as compared to the tournament barrier and MCS. Overall, the FLASH barrier execution time is 4% slower than the tournament barrier and 10% slower than MCS. We describe two factors that are likely contributors to this slow-down, arising from both application and FLASH system characteristics, though other application interactions may also be at work.

First, this version of Ocean uses aggressive prefetching, that accounts for between 50 –75% of all the read misses. These prefetches are used especially in the multigrid solver's red-black grid copy phases, which occur right after a barrier. Prefetches immediately after the barrier may not only impede the barrier itself, but also swamp the protocol processors and cause network queues to fill. The FLASH barrier's more uniform release characteristics exacerbate this contention and may cause performance to degrade as a result.

A second factor that may contribute to the performance differences is that the three barrier primitives release processors in a different order. The trees themselves not only have different structure, but even if they were the same, the releases would still differ greatly in their timing. Differences in processor communication patterns or bottlenecks are likely to be stimulated by this effect. Still, this effect does not seem to dominate, since the last arrival interval histograms indicate the final arrivals tend to have approximately the same characteristics across the different primitives, suggesting no single node dominates as the critical path.

Overall, we see that the FLASH barrier technique offers the ability to improve performance, as shown particularly in the 258x258 problem size that spends significant time in barriers. Ocean's communication patterns and other effects dominate in the 514x514 problem, even though the barrier metrics show the primitive's performance itself remains robust even in those circumstances. These indications reinforce our microbenchmark analysis that FLASH barriers are a powerful technique that offers the ability to synchronize at low cost, even at large machine sizes.

6.4 Discussion

This section reflects on the design of the synchronization primitives to identify their successes and propose approaches to address their limitations. Overall, we find that these protocols for FLASH provide an interesting counterpoint to the memory copy protocol we studied in Chapter 5. To provide coherent data movement at high performance, memory copy contained significant amount of protocol code and state. The synchronization primitives focus instead on achieving high performance through specialization and by striving to keep the protocol code small and efficient.

Locks

The FLASH lock protocol we describe represents only one of the many approaches to improving lock performance with specialized support. Our approach focuses closely on reducing artifactual communication for contended lock accesses. This choice turns out to be very effective, showing improvements for both Water and Barnes at large machine sizes. The performance improvements come not only from reducing lock latency itself, but even more importantly from the reduction in protocol processor occupancy artifactual communication causes.

On the other hand, consistent with the predictions and explanation of our microbenchmark analysis, we find that FLASH locks do not perform as well for non-contended accesses. In fact, at small application sizes where contention is low, we sometimes see a slight slowdown from the use of the FLASH technique. In those cases, FLASH locks increase the lock latency and occupancy since our handlers are longer and there is little or no artifactual communication to eliminate.

To address this limitation, future research in this area might consider a primitive that more effectively achieves the advantages of shared memory and FLASH locks. We suggest three approaches to such a hybrid that are natural extensions of the technique we present: *(i)* A hybrid solution where both shared memory and FLASH lock primitives are used, and are selected by the programmer using knowledge of the application characteristics. *(ii)* A dynamic hybrid primitive implemented in the lock library, such as reactive synchronization, where the lock shifts to a FLASH-based mode when contention is detected [LA94]. The back-off interval in the existing LL/SC primitives would allow this detection to be done at little or no impact when contention is not present. *(iii)* A dynamic hybrid primitive implemented in MAGIC, a unique opportunity provided by the flexibility in FLASH. This would operate similarly to the dynamic application-level primitive, falling back to shared memory in low contention, but by encapsulating the protocol in the memory system it is completely transparent to the processor. Assuming we adopt one of these solutions to address the low contention limitations of FLASH locks, the resulting primitive will have favorable characteristics over the broad spectrum of locking regimes. This may make FLASH locks a more resilient and stable protocol to use in general, especially in cases where a lock's contention behavior is not clearly understood, or cannot easily be anticipated.

We also note that our implementation encounters a number of limitations from allowing locks to be cached, such as those caused by speculation. Earlier we described the "Aggressive" lock variant, which attempts to mitigate somewhat the impact of speculation. More broadly, our application results find that caching of locks is not always critically important: in Barnes we find cached reacquisitions are fairly common, and help bring FLASH lock non-contended performance closer to that of the shared memory, while in Water its usage is less common. We might take advantage of this observation and consider a lock variant which *does not* support caching of locks. In such a primitive, the processor would communicate with the protocol processor via PPRs for *every* lock and unlock. This implementation would reduce the penalty associated with non-contended FLASH locks by eliminating two cache accesses in the critical path (one to read the lock state at the requester, and one to extract it from the current holder's cache) and would reduce the lock protocol's complexity and code size still further. Using a token-based approach, which we find provides significant benefits, this implementation would even permit local re-acquisition, though not at cache speeds.

Barriers

The barrier protocol builds upon conventional shared memory tree-barrier approaches by integrating a custom version into MAGIC. Our simulations across microbenchmarks and applications shows that based on the barrier metrics, this approach outperforms the other alternative in every case. Furthermore, even though application interference inevitably causes barrier performance to degrade somewhat as compared to the isolated predictions, the FLASH barrier protocol degrades much less than others.

On the other hand, our evaluation of FLASH barriers within the Ocean application shows mixed gains. In some cases, FLASH barriers do not improve overall execution time, probably due to other contention effects exposed in the application. Eliminating synchronization bottlenecks may always expose secondary bottlenecks of this sort. Other applications with less communication contention may show clearer gains from the improvement of barrier performance itself.

The barrier approach we choose is based on a static assignment of processes to processors. The internal MAGIC barrier tree requires reconfiguration if the mapping changes due to multiprogramming. While this is a restriction, we feel it matches the usage patterns of scientific applications, which use barriers the most frequently. Applications of that class generally attach to processors to improve locality. They also perform extensive memory placement operations to indicate their memory locality patterns; optimizing the barrier tree for processor assignment is a related optimization. In a commercial system environment in which attaching to processors is not permitted, the operating system would be employed to provide MAGIC with an updated processor assignment when mappings change, so the barrier tree can be updated to match.

6.5 Summary

This chapter presents two custom synchronization protocols for FLASH, locks and barriers. In each case, the design focuses on the low-level communication pattern the operation requires and identifies how conventional software shared memory implementations deviate from those patterns due to the cache coherence protocol.

The lock protocol targets in particular high contention situations where shared memory locks degrade significantly. Our protocol is based on a distributed queueing approach that permits locks to be handed directly from one holder to the next. It also permits caching of locks to allow re-acquisition without external communication. Our approach is similar in many respects to the QOLB primitive supported by SCI, but our primitive is more robust and is somewhat simpler. Our evaluation of FLASH locks using microbenchmarks shows that contended performance improves significantly as compared to non-queued locks, and gains about 40% over MCS locks, a software queued lock implementation, though non-contended performance is inferior to both alternatives. We also study applications taken from the SPLASH-2 benchmark suite, finding that FLASH locks enable continued scaling to 128 processors where speedup often drops off after 64 using conventional locks. These gains derive from superior locking as well as decreased controller occupancy arising from a reduction in communication artifacts.

The barrier protocol follows conventional software tree-barrier approaches, while implementing the operation entirely within MAGIC. Just as in locks, this approach reduces communication artifacts from contention to join the barrier, and improves performance by communicating directly between nodes. Microbenchmark analysis demonstrates that the join phase of the barrier outperforms even tree-based shared memory approaches by an order of magnitude under contention, and a factor of two to four when a single processor arrives late. The release performance is also significantly faster and thus more uniform, releasing 128 processors within 3 μs. Besides the normal barrier, we also demonstrate a variant which provides master-slave functionality, which is useful in some programs such as those generated by an auto-parallelizing compiler. Simulations using the Ocean application from the SPLASH-2 suite confirm the microbenchmark predictions of the barrier performance metrics, and further show that FLASH barriers maintain their performance under application interference better than conventional approaches. Execution time benefits are mixed: FLASH barriers improve overall performance for some cases, while others encounter secondary bottlenecks in the application.

The synchronization protocols benefit greatly from the flexibility of MAGIC overall, though two features are particularly useful. First, the presence of alternate address spaces allows the protocols to respond to cache misses for lock or barrier state with different handlers than for normal misses. This avoids the need to add special cases to the cache coherence protocol. Second, the flexibility of the Jump Table allows these specialized coherence handlers to be dispatched directly rather than through software, which would erode many of the latency gains the protocols achieve.

Chapter 7

Extensions and Future Directions

The flexibility provided by the unique design of MAGIC allows the FLASH system to support a wide range of functionality not possible in conventional systems. In the previous chapters, we have studied in detail several protocols that use this flexibility to accelerate specific kinds of communication operations. In this chapter, we describe a wide range of other protocols which may be promising directions of future research.

7.1 Active Messages

Chapter 5 describes the "block transfer" uses of message passing, focusing on the benefits it can provide for efficient communication. Some message passing programming models also provide the ability to invoke computation on a remote node when the message arrives there, e.g., `hsend` in NX [Pie88]. Note the contrast between this kind of "imperative" communication model and shared memory, which typically relies on the receiver explicitly polling to check if external communication has arrived.[1] Researchers have named this imperative style of communication *active messages* because of the computation such messages invoke upon receipt [vECGS92].

In this section, we describe the design space for active messages on FLASH. There are two issues which characterize our options for an active message implementation: The first is whether the computation expressed in the message should be executed on the main processor or on MAGIC's Protocol Processor. The second issue is whether to allow the user to provide arbitrary code to execute as an active message handler or only to allow system-level code.

[1]Some shared memory systems implementations, such as DASH [Len92], do provide a means to send inter-processor interrupts. What we describe here is a somewhat more advanced technique for invoking computation remotely.

7.1.1 Processor-Implemented Active Messages

The obvious starting point for active messages is to carry out the computation on the main processor. The main processor provides several advantages over the Protocol Processor. First, it is highly optimized for general code, more so than the PP which is geared towards stylized handlers with fairly mild computational requirements. Second, unlike the PP, the main processor provides floating point support in hardware. Finally, it provides the ability to run user code directly using hardware memory management features.

However, the strengths of the main processor are also to blame for its main weakness: invoking computation there has a high start-up cost. Through the processor interface, MAGIC controls a special bus that can interrupt the processor when needed. Unfortunately, deep pipelining, long-latency floating point operations, large register files (which must be saved and restored), and OS protection requirements all combine to make the interrupt latency very large in the R10000 (as in most modern high performance processors running conventional operating systems). The R10000 vectors to an interrupt handler 12–20 cycles after the interrupt is asserted. The software start-up cost may be as high as hundreds of cycles once the processor responds, depending on the operating system and whether or not a fast interrupt vector is reserved for this use.

This drawback aside, invoking active messages on the processor is a powerful technique to explore. One feature we will need to provide is a means of supplying the handler address to the main processor when a message arrives. In some processors which deal with messaging more directly, support for receiving a message and its handler PC is highly integrated with the processor [CSS$^+$91, DFK$^+$92, NPA92, ACD$^+$95]. In our case, we are forced to rely on the R10000's interrupt handler requesting the message from MAGIC when it is ready to service the active message. For performance, this interface should allow the processor to read the active message efficiently (i.e., cache it), but also unambiguously indicate that the message processing is complete (in case another message is waiting). These issues render a high-performance main-processor interface to active messages a challenging implementation. We describe our approach to requesting service on the processor in Section 4.1.3.

Protection Levels

Protection remains an important issue for the implementation of active messages. Active messages could execute either at user-level or kernel-level—each places protection burdens on the sending and receiving processors and MAGIC chips.

User-level handlers are the most flexible and widely usable since applications can provide their own code to run upon message arrival. Running handlers at user level is straightforward on the processor since it already provides the protection features that are required. However, user-level code must run in an address space appropriate for its application, which raises several issues:

- To provide this support, the processors and MAGIC need to communicate the *authenticated* identity of the sender so the appropriate address space can be selected. The operating system and MAGIC cooperate so MAGIC can provide the identity of the sender at low latency.
- Once the appropriate address space is selected, the OS must generate a context in which the handler can run. This may take a form similar to a `signal` handler (which uses the application's stack, thus suspending the application until the handler completes). If handlers are to be more general and run concurrently with the destination process, additional threads of activation must be ready in advance (or be prepared on the fly).
- The OS must verify the valid association of sender and receiver. This, combined with installing the correct address space further increases the handler-invocation latency above the hardware's lower bound.

We may also want to provide kernel-level active messages for inter-node kernel communication. Kernel active messages share the requirements of user-level ones, especially the requirement for authentication, to assure that only other kernels should be able to generate them.

7.1.2 MAGIC-Implemented Active Messages

Unlike previous systems that provided flexibility only in the compute processor, the Protocol Processor in MAGIC enables us to execute active message handlers in the memory system as well. This flexibility allows us to invoke handlers at significantly lower latency than on the main processor, and to provide the handler with more direct access to the network. On the other hand, the design of the PP places significant restrictions on the code that can be executed there (as described in Section 2.3.2).

System-level

The protection limitations of the Protocol Processor only permit it to execute *trusted* code, similar to system-level code on the processor. In other words, handlers may be invoked only by privileged users and must satisfy all the handler requirements of MAGIC. One form these system-level handlers may take is a set of available functions that are previously verified to be trusted. In a broader sense, the alternate protocols described in this thesis are examples of the functions that can be provided. If other handlers are desired, they must be verified in advance, since the PP provides little protection against errant handlers.

User-level

Despite the restrictions on handlers, there remain techniques which could be used to provide user-level handlers on the Protocol Processor. These techniques serve to assure protection for the system either statically, in advance, or at run-time. Though allowing user-level handlers to execute, the need to conservatively assure protection for the system is likely to reduce the performance of user-level handlers substantially. A more detailed evaluation of the performance and implementation of user-level active messages in FLASH is beyond the scope of this dissertation.

Compile-Time Verification Instead of writing protocol handler instructions directly (or through conventional compilation), they could be expressed in a specialized high-level language and converted by a custom compiler to protocol code. At the same time, this compiler could verify the code satisfies the requirements for correct handlers and only accesses valid data. To assure this correctness, the MAGIC features which provide direct hardware access would be expressed as calls in the high level language. In translating these calls, the compiler can insert the necessary correctness checks to prevent them being used inappropriately. For example, a `send` instruction would only be permitted if queue-space requirements were guaranteed by appropriate static invariants or coupled with a run-time queue-space check inserted by the compiler. Using a compiler to generate the protocol code has the advantage of utilizing the PP directly, but assuring the handler correctness requirements in a compiler is likely to be very difficult.

Run-Time Emulation and Sandboxing An alternative to specialized compilation is the reliance on exclusively dynamic checks for correctness. As an extreme solution, arbitrary user code could be *emulated* on the Protocol Processor, with the emulator performing protection checks where needed to ensure correctness. Emulation makes it is easier to guarantee valid handler execution, but does so at a significant performance cost. Most emulation systems slow down execution between a factor of 50 to several hundred as compared to native execution [DLHH94, RHWG95]. The slow down to emulate the PP is likely to be still worse because it normally executes two instructions per cycle. Despite this performance degradation, for short handlers performing basic tasks emulation may be faster than interrupting the compute processor.

The sandboxing technique proposed by Wahbe et al. [WLAG93] provides some of the features we require for direct user-level handler execution. Sandboxing entails rewriting the object code of an untrusted module so that loads and stores are very efficiently prevented from modifying memory outside an allowed region. Although preventing inappropriate memory accesses by user-level handlers is an important component of the protection we require, it still does not assure that handlers terminate or use MAGIC-resources appropriately. The environment described in [WLAG93] relies on a preemptive operating system and traditional operating system memory management features as a safety net in some cases. We would need to add runtime checks around potentially hazardous instructions like sends and ls/ss instructions, and also insert periodically executing code to allow the PP to regain control if the handler exceeds reasonable run time limits. Given those additional requirements, a sandboxing-like technique has the potential to greatly improve performance for user-level handlers running on the PP as compared to emulation or naive per-instruction checks.

7.2 Other Protocols

The protocols described previously are only a subset of the many possible ones that could be implemented on FLASH. In this section we briefly consider several of the other possible protocols to provide a broader picture of the features that MAGIC can support. Some of these protocols have already been studied by other researchers in the context of FLASH, while others are potential future research directions. A more detailed analysis of these protocols is beyond the scope of this dissertation.

7.2.1 Fetch-and-Op

As part of the NYU Ultracomputer and IBM Research Parallel Processor Prototype (RP3) projects, researchers developed a scalable synchronization primitive known as *Fetch-and-Add*, which serves as an alternative to synchronization based on locks [GLR83, GGK$^+$83, PBG$^+$85, FG91]. Fetch-and-Add provides a single atomic read-modify-write (an add) to a memory location. A generalization of Fetch-and-Add is *Fetch-and-Op* (also known as *Fetch-and-*Φ), in which *Op* can be essentially any math operation.

To explain the semantics of Fetch-and-Add, consider a memory location X with starting value 1000. Consider two processors executing `FetchAndAdd(X,1)` simultaneously. When the requests arrive at memory, one of the updates occurs first, but each occurs atomically. As a result, one of the processors updates location X to 1001 (receiving 1000 as its return value), the other updates it to its final value of 1002 (and receive 1001 as its result). Regardless of the interleaving, Fetch-and-Add assures that the final result is 1002 and that the two processors each receive unique return values 1000 and 1001. Non-atomic updates could mistakenly leave location X with the value 1001 and return value 1000 to both processors.

The implementation of Fetch-and-Op is attractive for high-contention situations since the operation is executed at the *memory*. If normal cacheable operations were used, the memory line

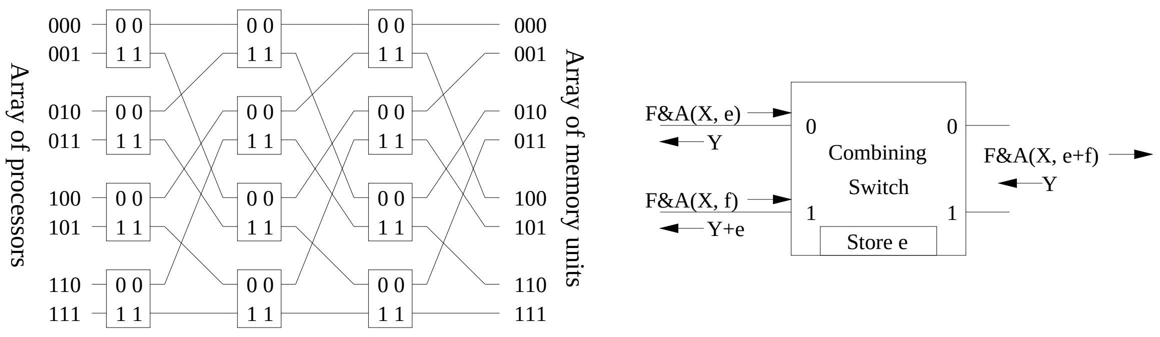


Figure 7.1: The NYU Ultracomputer's Combining network (taken from [GGK$^+$83]). Left: The dance-hall machine architecture with processors and memory elements separate and connected by an indirect Omega network. Right: the extensions to the network switch to provide the combining function.

would instead move around the system as the updates were performed, with each request experiencing long remote access penalties. In the Ultracomputer, in fact, the memory and a specially-designed interconnect called a *combining network* cooperate to accomplish this task still more efficiently [GGK$^+$83]. Figure 7.1 illustrates this network: the left panel shows a high-level view of the Omega network used to connect processors and memory. This type of system is commonly referred to as a *dance hall* architecture, since the processors and memory are each grouped together. The Ultracomputer extended the Omega network switch, shown at right, to provide the combining function. If the switch detects two Fetch-and-Add operations to the same location, it aggregates them into a single operation representing a superposition of the two and passes the combined operation onward. Later, when the result Y returns, the switch expands the operation to satisfy the two original requests.

In FLASH, Fetch-And-Op on integers is relatively easy to implement in the Protocol Processor. Since protocol handlers cannot be preempted, the PP handler merely needs to carry out a read-modify-write cycle to assure atomicity. Unfortunately, floating point operations would need to be emulated in PP software or executed directly on the compute processor (through interrupts). To estimate the performance of an IEEE 754 compliant floating point add in the Protocol Processor we studied the code generated by our compiler for a `gcc` math library function implemented in C. The latency of this operation varies widely between 20–100 cycles depending on the operands. Given the high cost of processor interrupts, this approach may still be faster than using the main processor.

7.2.2 Global Reduction Operations

Some massively parallel systems provide a special flavor of parallel communication, called *global reduction operations* (or simply *global ops*). These are similar to barriers since all processors rendezvous during the operation, but unlike barriers, global ops also compute a parallel arithmetic operation in the process. That is, each processor provides a numeric argument to the global op, and at the completion of the operation, each processor receives the result some function applied to the numbers provided by all the nodes. For example, in a global *add*, each processor provides a number $x_i, 0 \leq i \leq n$, where n is the number of nodes participating. As the return value from the global add, each processor is returned the sum of all the values $f_{0...n}$, where $f_{0...n} = \sum_{i=0}^{n} x_i$.

Global ops are provided by a variety of systems, e.g.: NX [Pie88] provides a range of global op functions such as `gdsum()`, MPI [Mes93] provides the `MPI_Reduce` function. FLASH, too, can provide global ops in the Protocol Processor, with the caveat that only integer operations can be implemented directly. Floating point would need to be emulated, as described in the discussion of Fetch-and-Op (Section 7.2.1).

7.2.3 Fault Containment, Reliability, and Recovery

In a collaborative effort between the system designers and the operating system group, FLASH researchers have been designing support for reliability in the system. This design uses two major cooperating thrusts: a scalable fault-containing operating system, and techniques for fault detection, containment, and recovery through support from custom MAGIC protocol code.

***Hive* Operating System** The operating system group of the FLASH project is designing a new UNIX operating system called *Hive* [CRD$^+$95]. Hive extends the commercial IRIX operating system from Silicon Graphics to improve scalability and reliability and to increase its awareness of the machine's NUMA characteristics. To improve reliability, with the aid of the hardware support described below, Hive divides the system into fault containment boundaries called cells. Hive also leverages fast active message support for kernel communication between cells, each of which runs their own copy of the kernel.

Fault Detection, Containment and Recovery Underlying the Hive system are advanced protocol features that: *detect faults* that occur, *prevent faults* from affecting other processors or cells, and *recover* data or nodes corrupted because of faults [TBG$^+$97]. Collectively known as the *recovery protocol*, these handlers ensure the system's integrity by using the special handler MAGIC executes periodically, called the *idle handler* to check for problems. If error conditions are detected, the handlers bring the system to a quiescent state and explore the system to determine the extent of the fault's effect. The protocol then restores to full operation the viable portion of the system, and isolates and disables nodes that have experienced hard failures.

7.2.4 Performance Monitoring

One of the many beneficial uses of the flexibility in MAGIC is to export performance monitoring features to the system. Performance monitoring can be used to study the system at several different levels. To the designers of FLASH, performance monitoring is useful to determine how efficiently the MAGIC chip is functioning. This feedback can enable the designers to customize some programmable parameters in the chip to optimize its performance, to determine if software or hardware errors are occurring, and to examine whether bottlenecks exist in the current design. Performance monitoring of this sort is provided in two ways. Hardware counters embedded in MAGIC provide a view of hardware events too small or too difficult to measure in PP software. Instrumented handler code (selected using a special sampling feature of the Inbox Jump Table) is used to read these counters at appropriate times and measure higher-level events like protocol caching effectiveness.

At another level, performance monitoring can be used by the application programmer to study not the system, but rather the performance of the application running on it. In the past, tools such as MemSpy [Mar93] have relied on program simulation or run-time statistical sampling to determine program behavior. Some processors, such as the Intel Pentium also provide a counter-based facility that can be used to provide simple performance statistics. Binary rewriting tools such as pixie for MIPS systems and similar tools [SCH$^+$91] instrument executables to provide exact profiling, though they perturb application execution. In FLASH, performance monitoring can be provided much more powerfully through support from MAGIC, including the visibility of details not available through any existing techniques. The FlashPoint protocol [MOH96] implements performance monitoring in this style by extending the coherence protocol to maintain additional information about memory system behavior. Verghese et al. [VDGR96] also use MAGIC to feed memory access statistics to the operating system to guide page migration and replication decisions. Unlike FlashPoint which merely provides the programmer with information useful for tuning the application by hand, in this case performance monitoring may allow the kernel to dynamically improve application performance. The ability to provide protocols such as FlashPoint, to dynamically influence page migration, and to supply the programmer or operating system with real-time feedback is a powerful advantage of FLASH.

7.3 Summary

This section described a range of other alternate protocols that may be interesting research direction for FLASH or similar machines. We described active messages, an alternative communication style that shows promise for an efficient implementation with support from MAGIC. Our discussion focused on the range of design choices available in that protocol and to its users.

We identified two memory access primitives with unique characteristics targeted at supporting different contended scenarios. Fetch-And-Op is designed for simple manipulations under high-contention, while global ops are used to export all-to-all communication primitives using the facilities of the machine as efficiently as possible. Both of these primitives are likely to perform well on FLASH and may benefit from the implementation techniques used for synchronization primitives.

We also identified a few other interesting uses of the embedded flexibility of MAGIC. Operating systems can leverage this flexibility to increase performance, reliability, and scalability. System designers and application programmers can use the increased visibility into the memory system to better understand the performance tradeoffs in their designs.

Chapter 8

Conclusions

A programmable protocol engine provides a novel and powerful model for supporting many classes of communication within a multiprocessor. This research studies one such environment in particular, the FLASH Multiprocessor, which was designed from the beginning with embedded flexibility in the memory system. Complementing previous studies of FLASH focusing on cache coherence, this dissertation presents an analysis of issues for other protocols FLASH can support.

Our study shows that the programmable protocol engine in FLASH is able to effectively support protocols such as block transfer and synchronization using the same hardware provided for implementing cache coherence. This is due in large part to the flexible design of the MAGIC node controller, which provides an optimized programmable protocol engine and generalized communication mechanisms that efficiently move data in parallel. It is also a result of the careful design of the protocols themselves, which identify the critical limitations in conventional implementations and leverage the node controller's support to address them.

8.1 Interface Between Processor and Controller

One of the major issues in dividing communication functionality between the processor and controller is effectively supporting their interaction. Interfaces with high overhead or functionality restrictions can limit the ability of alternative protocols on MAGIC to improve performance or cooperate with processor applications. Addressing the limitations of prior research, our approaches enable processor-controller interaction at low overhead while also supporting modern operating system requirements.

In the first part of this interface we describe techniques to provide efficient, reliable communication between the processor and MAGIC, such as the ability to invoke alternate protocol functions. The main innovation of this approach is the ability to reliably communicate with MAGIC at user level without system calls, while assuring protection and atomicity even in the presence of multiprogramming. We find that our specially designed memory-mapped interface is a convenient and powerful approach since it builds on the processor's existing memory interface and requires only minor support from the operating system. We provide a related technique for communication from MAGIC to the processor using interrupts, allowing the processor to service critical requests generated by protocol handlers.

Second, we present novel techniques to enable MAGIC to cooperate with the virtual memory facility of the processor operating system. This requirement is unique to our study since the cache coherence protocol operates entirely using physical addresses, while many classes of alternate protocols must support virtual addresses. Our techniques enable reliable virtual address communication from the processor to MAGIC and also protect the integrity of those addresses while they are in use, should paging cause the mappings to change. Though MAGIC lacks a hardware-based TLB due to design complexity tradeoffs, we demonstrate how software approaches including a software TLB can efficiently support similar functionality.

The combination of user-level access to the communication controller and support for protection and virtual memory is unique feature of the FLASH architecture, enabled by the flexibility of the node controller. By providing these features for efficient operating system coexistence, we find the alternate protocols we study for MAGIC can be used effectively by conventional applications with few or no restrictions.

8.2 Memory Copy

Our first detailed protocol study considers memory copy functionality, in which the processor delegates the transfer to MAGIC. The controller performs the transfer in the background, enabling parallel computation and communication. Our study of memory copy identifies conclusions in two areas in particular:

First, unlike systems which provide custom support for block transfer, our protocol shares the same hardware as cache coherence and thus MAGIC transfers the data to the destination processor a line at a time. On the one hand, we find that sending lines individually requires careful protocol design to achieve high performance. We show how techniques such as software pipelining and software-controlled speculative execution are invaluable to optimize data transfer. We also observe that transfers using this design must have a common cache-line alignment between source and destination buffers for peak performance. Changing data alignment is expensive in software, motivating the addition of a simple hardware feature in FLASH to realign data as it is loaded into data buffers (the only additional feature provided for memory copy). Using this feature, unaligned transfers are very efficient, though they are still 25–50% slower than aligned transfers due to additional protocol processing needs.

On the other hand, sending lines individually and at the same granularity managed by the coherence protocol provides a number of useful benefits, especially given our approach of providing complete integration with cache coherence. First, sending individual lines allows increased parallelism that is valuable to hide remote communication latency such as that caused when data needed by the transfer is found to be cached remotely. It also enables more aggressive protocol designs, which we explore in detail, unlike prior approaches which rely on first flushing the involved data from all the caches in the system.

Second, many prior systems have provided a transfer model with restricted coherence support called *local coherence*, arguing that it reduces complexity, improves performance, and optimizes the common case. Our observations suggest that remote caching is prevalent and that only a fully-integrated protocol can be readily used within shared memory applications, including the operating system, without restrictions. Our implementation supports the complete generality provided by a cache-coherent shared memory system in which data may be cached remotely, and may be allocated from remote memory. The results of our study also suggest that providing full coherence support comes at a fairly low cost. Though the additional work required to handle arbitrary caching of message data decreases transfer performance when used, the presence of this support does not noticeably slow the local caching cases and thus there is little motivation to restrict the implementation to just local coherence.

We evaluate the memory copy primitive using microbenchmarks that isolate its usage within shared memory or message passing programs, and an application highlighting block transfer use by the operating system. We find that block transfer in MAGIC provides some advantages including improving overall performance in certain cases. In other cases processor copy with prefetching can match the raw performance, but does so at a cost of occupying the processor and polluting its cache, both of which the FLASH protocol avoids. While gains from the protocol are small to modest in FLASH due to its relatively low remote memory latency, the potential benefits stand to be much greater in systems with longer remote latencies, or where protocol processing throughput improves from hardware or software optimizations.

8.3 Synchronization

We also study lock and barrier synchronization primitives using custom protocol support from MAGIC. These protocols are motivated by the observation that synchronization primitives implemented on top of shared memory incur significant wasted communication due to cache coherence, especially under contention. By targeting this “artifactual” communication, our protocols improve

Table 8.1: Protocol code size summary. The cache coherence protocol is included as a comparison point.

Protocol	Code Size
Memory Copy	19 KB
Locks	2.8 KB
Barriers	2.2 KB
Cache Coherence (dynamic pointers)	44 KB

contended synchronization performance and also decrease the impact of high contention cases on the rest of the system.

Our lock protocol takes an aggressive approach utilizing distributed queueing, enabling a lock to be transferred from one holder to the next using a single FLASH message. Unlike previous protocols that use distributed queueing for cache coherence, such as SCI which maintains distributed doubly-linked lists, our approach is simple, efficient, and is specialized to perform a single function well. Our protocol is also more resilient to exceptional conditions that arise and is more robust at assuring fairness than some prior approaches such as QOLB, because we maintain state within MAGIC and not processor caches. One limitation to our approach is that it cannot easily support collocation of locks and application data, a feature enabled by protocols such as QOLB that are more closely linked with cache coherence. Collocation is most useful when locks protect small data structures that can fit on the same cache line, thus its potential benefits are lower for the typically larger lock-protected data structures in the applications we study. Our protocol is also less appropriate for low-contention cases, since shared memory performs very well in that regime. We propose a range of future research to address that limitation through protocol modifications or hybrid implementations.

The barrier protocol is similar to optimized shared memory approaches except that its tree communicates between the MAGIC chips on different nodes. Our results show unequivocal gains from this implementation, demonstrating that the additional communication between the processor and MAGIC in conventional barriers amounts to significant overhead. Though barriers are somewhat less prevalent than locks in parallel applications, reducing their cost may encourage their more widespread use. Furthermore, research in areas such as auto-parallelizing compilers suggests that the ability to synchronize processors at extremely low overhead will become increasingly valuable; the FLASH barrier variant customized for that class of applications shows similar improvements over conventional techniques.

One secondary attribute of both synchronization protocol implementations is that they are incredibly compact, fostering improved MAGIC instruction cache sharing with the major cache coherence protocol running on the machine. The cache coherence and memory copy protocols are significantly larger—as much as an order of magnitude. Table 8.1 summarizes the code sizes of the protocols we study and the cache coherence protocol (for comparison). This compact synchronization protocols are made possible by *(i)* specializing each protocol to provide only a specific operation, *(ii)* providing a simple interface to communicate with the protocol, and *(iii)* carefully designing the protocol to reduce the code explosion from local and remote interactions. While not all protocols can utilize these optimizations, it demonstrates that useful protocols for MAGIC can be very compact in some cases.

8.4 Flexible Controller Design Observations

Previous studies of FLASH focus predominantly on the features and limitations which impact cache-coherent shared memory protocols. One benefit of our study is that it considers the FLASH design, and in particular the microarchitecture of MAGIC, with an eye towards a very different class of usage. This section reflects on several design issues for flexible controllers, integrating together the observations from across the different parts of our study.

One area with significant impact on our protocols is the processor itself and the processor interface. In general, our protocols benefit from flexible control over the manipulation of data, especially the manipulation of processor caching. Paradoxically, our protocols' needs for controlling the cache are more advanced than those of the cache coherence protocol, largely because we perform more complex operations such as block transfer. The R10000 and the MAGIC processor interface support this effort by allowing two outstanding cache extraction interventions. Though the R10000 supports even more, we find two requests to be sufficient for performance and that the complexity of managing more requests is prohibitive anyway.

On the downside, the R10000 interface has two particular limitations. First, it does not allow updates to be "pushed" into the cache, even for data which it currently holds. This increases the overhead and complexity of some cases, such as updating cached lock and barrier state, which instead invalidate the cache and yield the PP to allow the spinning processor to miss. In block transfer, it also reduces the protocol's ability to supply data to the processor in anticipation of its use, which is one reason prefetching can sometimes outperform our block transfer protocol. Second, the R10000 prevents the user from flushing lines out of the cache, restricting this operation to privileged levels only. This limitation, combined with the presence of processor speculation, forces the processor/MAGIC communication techniques to use uncached accesses, which are less efficient. The inability to flush lines also prevents the processor from writing back data in anticipation of an upcoming block transfer, which would improve its performance. These two restrictions are not surprising given the needs of most coherence protocols and microprocessor applications, but they are unfortunate from the perspective of the protocols we study.

Another way in which our protocols tax MAGIC more than cache coherence is our heavy usage of data buffers. For example, we find great benefits in our block transfer protocol from the ability to software pipeline the main transfer handler. This amortizes transfer overhead across multiple lines and hides processor cache access latency behind other processing. MAGIC's general data buffer mechanism and the protocol processor's ability to control data movement manually are necessary to enable these optimizations. Both of these features are in contrast with more restrictive data buffer approaches used by recent commercial systems similar to FLASH.

One significant design tradeoff related to data buffers is the tracking of outstanding requests. The Sequent NUMA-Q system stores a "handler continuation" for requests that are launched, which resumes when the associated reply arrives. MAGIC uses a "fire-and-forget" approach where no state is kept for lines once they are launched. Aside from the fault tolerance implications we do not address, this has a clear performance tradeoff. The continuation-based approach reduces overall latency by allowing the reply to be processed immediately at the point in the handler after the initial send. In contrast, MAGIC is forced to dispatch a new handler for the reply message and (typically) analyze state to determine how to process it. Despite this drawback, the continuation-based approach necessarily imposes fundamental limits on the number of outstanding messages due to practical hardware constraints. Processor prefetching has similar limitations. We find the ability to have many block transfer lines in flight invaluable for hiding latency, an effect which was also observed by other researchers, and will be amplified as effective network latencies grow.

Another important feature of MAGIC for our protocols is the flexibility afforded by the programmable handler dispatch table, the "Jump Table". The Jump Table allows handlers to be dispatched using a number of criteria, including address spaces and uncached read and write flavors. Our protocols rely heavily on those distinguishing marks to cue special handling within MAGIC. For example, the ability to launch a separate coherence protocol using a different address space was integral to the synchronization protocols we study. Though software dispatch could always be used, its performance is significantly slower; the generality in the Jump Table combined with careful message encoding enabled us to completely avoid software dispatch in all common case handlers.

Given our aggressive use of programmability and the wide range of characteristics in the protocols we study, it is predictable that we should advocate generality within the mechanisms in MAGIC. From a practical perspective, however, the needs for flexibility must be tempered with the requirements to achieve high performance for machine common cases like cache coherence. Our experience suggests that MAGIC achieves a surprisingly effective balance between these two conflicting goals, allowing a wide range of communication types to be supported in a single system.

Appendix A

MAGIC Implementation Details

This appendix extends the basic description of MAGIC in Chapter 2, providing some lower-level detail about several parts of the design.

A.1 PP Instruction Set

As described in Section 2.3.2, the core instruction set of the PP is based loosely on that of the MIPS R3000, with extensions to improve performance of common protocol operations. Table A.1 summarizes the instruction set of the protocol processor. Most of the instructions are self-explanatory; the data buffer operations (lblock, sblock) are described in more detail in the discussion of the data buffers in Section 2.3.4.

A.2 Processor Interface

Section 2.3.3 describes the processor interface briefly. We expand on this initial description to clarify the function of the PI Reply Register and explain the kinds of operations the PI can support.

A.2.1 PI Reply Register

Since the R10000 controls its own second-level cache, the PI handles interventions by issuing the request on the bus and then waiting for a response. This response time from an intervention is variable because the processor may be in the middle of a long-latency operation that needs to complete before it can answer. In fact, the PP may or may not be interested in the response code provided by the processor. This arises because some requests (such as invalidations), always succeed and

Table A.1: Summary of the MAGIC Protocol Processor instruction set.

Instructions	Instruction Class	Description
add, sub, and, or, nor, xor	ALU Ops	Standard ALU operations
sll, sllv, srl, srlv	Shifts	Shifts, including support for shifts of more than 32 bits
*op*i, *op*fi, *op*ifi	Bitfield operations	Generate a mask of contiguous bits and combine it with a register operand, (*op* can be add, and, or, xor). e.g.: `andfi $1, $2, 62, 63` — A mask of two high bits is anded with $2 and stored in $1.
insfi, insifi	Bitfield insertion	Insert data into a contiguous bitfield in a register
ffsb	Bit vector	Find first set bit
j, jr, jal	Jumps	Unconditional constant and register jumps; jump and link
beq, bne	Fast-compare branch	Branch on equality/inequality of registers
blez, bgtz, bltz, bgez	Zero-compare branch	Branch on inequality comparisons against zero
bbs, bbc	Bit testing branches	Branch on bits set or clear
ld, sd	Load/Store	Load or store doublewords (only doublewords supported)
ls, ss	MAGIC state access	Load or store internal MAGIC state variables
send	Message Send	Send a message to the processor, network, or IO
switch/ldctxt	Context Switch	Load the two portions of the next message into registers, and jump to the next handler entry point
inv, copybk, ltag, stag	MAGIC data cache maintenance	Explicitly invalidate or copy back lines from the MAGIC data cache, or manipulate tags directly.
lblock/sblock	Data buffer memory operations	Explicitly fill or spill a data buffer, either with a single double word or a full cache line of data.

so the protocol handler need not consult the response. Other requests may fail (such as a request for a particular address in the cache if the line is not found), and so the protocol must wait for the response to see if the requested data can be delivered. The PP indicates its intention to the PI using a flag in the message. If the PP indicates that it is waiting for the response, the PI writes a reserved location called the *PI Reply Register* with the result code.

The PP can have only two outstanding requests for which it has requested a response. The PI Reply Register acts as a special kind of response queue: once the response is read, it frees the register to indicate the result of the next response. Thus, no special handling is required to read the *next* request's status reply: each request's result arrives the order they were issued. Instead, the protocol must take care to read the PI Reply Register exactly once for each request.

Table A.2: Explanation of Processor Interface interventions (partial list).

Op type	Description
Get	Request a copy of a line suspected to be in the cache. If the line is held in exclusive mode, return a shared copy of the data, leaving the line in the cache in the shared state. The state reply indicates the state (shared, exclusive, invalid/not-present) of the line when the request arrived.
GetX	Request a copy of a line suspected to be in the cache. Return the data and remove the line from the cache entirely. The state reply indicates the state (shared, exclusive, invalid/not-present) of the line when the request arrived.
Inval	Invalidate a line in the cache (regardless of its caching state), and do not return the data. The state reply indicates the state of the line when the request arrived. Note: Under normal circumstances, this request should never be issued for a line which is in the cache in the exclusive state, since that copy is the most up-to-date copy of the line. However, we describe a scenario in our message passing protocol in Chapter 5 in which we can optimize our protocol by invalidating exclusive lines in particular circumstances.
Put	Reply to a processor request, installing the line in the shared state. A Put may only be issued for a line previously requested by the processor. No state reply is provided.
PutX	Reply to a processor request, installing the line in the exclusive state. A PutX may only be issued for a line previously requested by the processor. No state reply is provided. Note: if the processor requested the line in the shared state (i.e., with a Get), it is legal to reply to the processor with a line in the exclusive state (i.e., with a PutX). We leverage this feature to optimize the lock protocol in Chapter 6
NAK	**N**egatively **A**c**K**nowledge a processor request. This may cause the processor to reissue the request, depending on whether the original request was speculative and later determined to be unnecessary. The use of NAKs is critical for deadlock avoidance in case node resources are temporarily insufficient [LLG$^+$90, LLG$^+$92, KOH$^+$94]. No state reply is provided.

A.2.2 Supported PI Operations

The PI provides a wide range of operations on data in the processor's cache, all of which find utility in the protocols we describe. Table A.2 briefly describes the semantics of these operations. In the table, we describe the processor's response to the request when sent by MAGIC *to* the PI. Similarly named requests *from* the processor are requests for the related coherence operation from MAGIC.

Appendix B

Synchronization Primitive Implementations

This appendix presents the pseudo code for the conventional lock and barrier primitives we compare against, as a reference for understanding their implementation-specific effects we encounter in our simulations.

B.1 Locks

We begin with the lock implementations we study in Section 6.1.2. Figure B.1 shows the LL/SC-based lock. The lock is first read using an LL operation. This load may find the lock held elsewhere, and if so it repeats the load (spinning in its cache) until the lock is again available. Then the processor tries to assert the lock itself and update memory using an SC operation. Failed attempts due to contention are addressed using exponential back-off in an attempt to decrease contention in successive rounds.

Figure B.2 shows the MCS lock, which improves performance over LL/SC through queueing [MCS91a]. We use the version of MCS locks based on the atomic store primitives Fetch-And-Store and Compare-And-Swap, which we internally implement with LL/SC. Fetch-and-Store works as follows: given an address and a value, it atomically reads the address, writes the supplied new value, and returns the old value. Compare-and-Swap is a conditional atomic store: given an address, an “old” value, and a “new” value, it atomically reads the address, compares the result to the “old” value, and if they match, stores the “new” value in memory, otherwise leaves memory unchanged. It returns a constant 1 or 0 to indicate whether the store occurred.

```
Lock()
{
 TryLock:

   lock = LoadLinked (lockAddr);

   if (lock)            /* Lock locked elsewhere */
      goto TryLock;

   lock = 1;            /* Try to acquire lock */

   result = StoreConditional (lockAddr, lock);

   if (!result) {       /* Contention for lock */
      ExpBackOff();     /* Wait a random amount */
      goto TryLock;
   }
   /* I now hold the lock */
}
```

Figure B.1: Pseudo code for a load-linked/store-conditional-based lock implementation.

```
typedef struct MCSnode
{
   struct MCSnode *next;
   volatile int locked;
} MCSnode;

typedef MCSnode *MCSlock;

/* L points to the shared lock */
/* I points to the requester's local queue entry */
AcquireMCSlock(MCSlock L, MCSlock I)
{
   I->next = NULL;              /* Initially I has no successor */
   pred = FetchAndStore(L, I);  /* Make I new tail, return old */

   if (pred) {                  /* Lock is not free */
      I->locked = 1;            /* Indicate lock is not free */
      pred->next = I;           /* Enqueue myself */
      while (I->locked)         /* Spin on lock */
        continue;
   }
}

ReleaseMCSlock(MCSlock L, MCSlock I)
{
   if (I->next == NULL) {       /* Currently no successor */
      if (CompareAndSwap(L,  I, NULL)) {
         return;                /* Still no successor, lock free */
      }
      while (I->next == NULL)
         continue;              /* Wait for successor to enqueue */
   }
   I->next->locked = NULL;      /* Release successor */
}
```

Figure B.2: Pseudo code for the Mellor-Crummey Scott (MCS) lock implementation, adapted from [MCS91a].

```
typedef struct {
   int count;
   int padding[];              /* Pad to new cache line */
   int generation;
} barrier_t;

Barrier(barrier_t *barrier, int num_procs)
{
   int gen = barrier->generation;

 loop:
   count = LoadLinked(&barrier->count);
   count++;
   if (count == num_procs)
      count = 0;

   result = StoreConditional (&barrier->count, count);
   if (!result)
      goto loop;

   if (count == 0) {
      barrier->generation = gen+1;
      return;
   }
   while (gen == barrier->generation)
      continue;
}
```

Figure B.3: Pseudo code for the LL/SC-based barrier implementation.

B.2 Barriers

This section describes the barrier primitives we studied in Section 6.2.2. The basic LL/SC barrier implementation is shown in Figure B.3. This barrier operates in a similar manner to the LL/SC lock primitive in that it updates a shared variable under mutual exclusion.

Figure B.4 presents the tournament tree barrier we study, implemented by Chris Holt. It uses a tree represented by an array of structures, in which a tree node's parent and children are calculated based on offsets into the array. The algorithm to manipulate an array-packed tree data structure is well-known and is abstracted here for brevity. Each tree node rendezvous and release is performed by manipulating a lock-protected count. Unlike the originally proposed tournament barrier, processors statically know which will advance up the tree based on their processor number.

The MCS barrier implementation is shown in FigureB.5 [MCS91a]. At a low level, this primitive differs from the tournament barrier in several ways besides those mentioned in Section 6.2.2. First, unlike the tournament barrier approach described above, MCS uses a pointer-connected set of tree nodes instead of a packed array structure. By pre-calculating the tree linkage when the barrier is initialized, MCS barriers eliminate tree position calculations when the barrier executes. MCS barriers also use a toggling barrier sense indicator so that each iteration of the barrier does not need to reinitialize the release tree.

```
struct {
  lock_t lock;                /* Assure atomic flag increment */
  int flag;
  int padding;                /* Padding to reach page size */
} PaddedFlag;

PaddedFlag *joinTree;
PaddedFlag *releaseTree;

inline
waitAndClearFlag(PaddedFlags *tree, int offset, int count)
{
   while (tree[offset]->flag < count)
      ;                       /* Wait for others' signal */
   clear tree[offset]->flag;  /* Initialize for next barrier */
}

inline
incrFlag(PaddedFlags *tree, int offset)
{
   LOCK(tree[offset]->lock);
   tree[offset]->flag++;
   UNLOCK(tree[offset]->lock);
}

/* Tree expressions used below
 *
 * my_parent:   my parent's offset in the tree
 * my_children: my children's rendezvous for this tree level
 */
TournamentTreeBarrier(int ProcId, int num_procs)
{
   if (leafNode) {
      incrFlag(joinTree, my_parent)
      waitAndClearFlag(releaseTree, ProcId, 1);
                              /* Wait to be released */
   }

   while (I advance up the tree) {
      waitAndClearFlag(joinTree, my_children, JOIN_RADIX);
      Advance level and update position in tree;

      if (I will not advance further) {
         incrFlag(joinTree, my_parent);
         waitAndClearFlag(releaseTree, ProcId, 1);
                              /* Wait to be released */
      }
   }

   if (tree root) {           /* Wait for final round */
      waitAndClearFlag(joinTree, my_children, JOIN_RADIX);
   }

   while (I have children) {
      incrFlag(releaseTree, my_children);
                              /* Release my children */
      Advance level and update position in tree;
   }
}
```

Figure B.4: Pseudo code for the tournament tree barrier implementation.

```
typedef struct {
   volatile int wsense;
   int *parentPointer;
   int *childPointers[MCS_LEAVE_RADIX];
   int haveChild[MCS_JOIN_RADIX];
   volatile int cnotReady[MCS_JOIN_RADIX];
   int dummy;
} MCSTreeNode_t

/* Tree expressions used below
 *
 * child(i, radix, num_procs)
 *   Pointer to my i-th child in tree of specified radix with
 *   maximum node count num_procs, or zero if no such child.
 *
 * parent(radix, num_procs)
 *   Pointer to my parent in tree of specified radix with maximum
 *   node count num_procs or zero if the root.
 */

InitMCSBarrier(int procId, int num_procs)
{
   MCSTreeNode_t *node = nodes[procId];
   node->wsense = 0;

   /* Construct join tree */
   for (j=0...MCS_JOIN_RADIX) {
      node->haveChild[j] = node->cnotReady[j] =
         (child(j, MCS_JOIN_RADIX, num_procs) != 0);
   }
   node->parentPointer =
      & parent(MCS_JOIN_RADIX, num_procs)->cnotReady;

   /* Construct release tree */
   for (j= 0 ...MCS_LEAVE_RADIX) {
      node->childPointers[j] =
         & child(j, MCS_LEAVE_RADIX, num_procs)->wsense;
   }
}

MCSBarrier(int procId, int *sense)
{
   MCSTreeNode_t *node = nodes[procId];

   /* Wait for children to join */
   repeat until node->cnotReady[0 ... MCS_JOIN_RADIX] all zero;

   /* Initialize join tree for next time */
   node->cnotReady[0 ... JOIN_RADIX] =
      node->haveChild[0 ... JOIN_RADIX];

   *(node->parentPointer) = 0;  /* Join with my parent */
   if (procId != 0) {
      while (*sense != node->wsense)
        continue;               /* Spin locally on sense flag  */
   }
   *(node->childPointers[0 ... MCS_LEAVE_RADIX]) = *sense;
                                /* Release my children */
   *sense = ! *sense;           /* Flip sense for next time */
}
```

Figure B.5: Pseudo code for the Mellor-Crummey Scott (MCS) barrier implementation, adapted from [MCS91a].

```
typedef struct MasterSlaveBarrier {
   volatile int entered[MAXPROCS];
   char _pad0[];                /* Pad to cache line */
   volatile int genNumber;
   char _pad1[];                /* Pad to cache line */
   int copyGenNumber;
   char _pad2[];                /* Pad to cache line */
} MasterSlaveBarrier;

/* Used by slaves joining the barrier */
MSBarrier_SlaveEnter(MasterSlaveBarrier *b, int myid)
{
   int gen = b->genNumber;

   b->entered[myid] = 1;
   while (gen == b->genNumber)
      continue;
}

/* Used by the master to join the barrier */
MSBarrier_MasterEnter(MasterSlaveBarrier *b, int numProcs)
{
   int i;

   for (i=1; i<numProcs; i++) {
      while (!b->entered[i])
         continue;
   }
   bzero(b->entered, numProcs * sizeof(int))
   /* Master falls through while slaves continue waiting */
}

/* Used by the master to release the slave waiters once
   the master’s coordination processing is finished */
MSBarrier_Release(MasterSlaveBarrier *b)
{
   b->copyGenNumber++;
   b->genNumber = b->copyGenNumber;
}
```

Figure B.6: Pseudo code for the Basic Master-Slave barrier implementation

Finally we show the basic master-slave barrier implementation in Figure B.6. This implementation is similar to the functionality of a single tree node in the MCS barrier, since it uses an array of flags (called `entered`) packed tightly together.

Bibliography

[AAG+87] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica Lam, Onat Menzilcioglu, and Jon A. Webb. The Warp computer: architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

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

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

[ACD+91] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large scale distributed-memory multiprocessor. Technical Report MIT/LCS Memo TM-454, Massachusetts Institute of Technology, 1991.

[ACD+95] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large scale distributed-memory multiprocessor. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[AGGW92] Nagi M. Aboulenein, Stein Gjessing, James R. Goodman, and Philip J. Woest. Hardware support for synchronization in the scalable coherent interface (SCI). Computer

Sciences Department Tech Report 1117, University of Wisconsin, Madison, November 1992.

[ALK$^+$91] Anant Agarwal, Beng-Hong Lim, David Kranz, et al. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[And89] Thomas E. Anderson. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 170–174, Aug 1989. Volume 2.

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

[AS88] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–24, August 1988.

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[BAM$^+$96] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[BAR96] Edouard Bugnion, Jennifer M. Anderson, and Mendel Rosenblum. Using SimOS to characterize and optimize auto-parallelized SUIF applications. In *Proceedings of the First SUIF Compiler Workshop*, January 1996. Stanford University, `http://www-suif.stanford.edu/suifconf/suifconf1`.

[BDFL96] Matthias Blumrich, Cezary Dubnicki, Edward W. Felten, and Kai Li. Protected, user-level DMA in the SHRIMP network interface. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 154–165, February 1996.

[Bec92] Michael J. Beckerle. An overview of the START(*T) computer system. Motorola Technical Report MCRC-TR-28, Motorola, Inc., One Kendall Square, Building 200, Cambridge, MA 02139, Jul 1992.

[BI86] Eugene D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, August 1986.

[Bla90] Tom Blank. The MasPar MP-1 architecture. In *Proceedings of COMPCON Spring '90: Thirty Fifth IEEE Computer Society International Conference*, pages 20–24, Mar 1990.

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

[Bra77] Achi Brant. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, 1977.

[BRG$^+$89] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation lookaside buffer consistency: A software approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, April 1989.

[BSSD96] David L. Black, Randall D. Smith, Steven J. Sears, and Randall W. Dean. FLIPC: A low latency messaging system for distributed real time environments. In *1996 USENIX Technical Conference*, pages 229–238, January 1996.

[BZS93] Brian Bershad, Matthew Zekauskas, and Wayne Sawdon. The Midway distributed shared memory system. In *Proceedings of COMPCON'93*, pages 528–537, February 1993.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CHRG95] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta. Memory system performance of unix on CC-NUMA multiprocessors. In *Proceedings of the 1995 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95/PERFORMANCE '95)*, 1995.

[Cra93] Cray Research, Inc. *Cray T3D System Architecture*, 1993.

[CRD$^+$95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 12–25, December 1995.

[CSS$^+$91] David E. Culler, Anurag Sah, Klaus Erik Schauser, et al. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, April 1991.

[DCF$^+$89] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, and Scott Wills. The J-Machine: A fine-grain concurrent computer. In *IFIP Congress*, August 1989.

[DFK$^+$92] William Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, 1992.

[DLHH94] Peter Davies, Phillipe Lacroute, John Heinlein, and Mark A. Horowitz. Mable: A technique for efficient machine simulation. Technical Report CSL-TR-94-636, Stanford University, Computer Systems Laboratory, Sep 1994.

[DOSW96] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. A message passing standard for mpp and workstations. *Communications of the ACM*, 36(9):84–90, July 1996.

[FAB$^+$96] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Liviu Iftode, and Kai Li. Early experience with message-passing on the shrimp multicomputer. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 296–307, May 1996.

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

[FLR$^+$94] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ionnas Shoinas, Mark D. Hill, James R.Larus, Anne Rogers, and David A. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercompting 94*, Nov 1994.

[Fly66] Michael J. Flynn. Very high-speed computers. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[FV93] Matthew I. Frank and Mary K. Vernon. A hybrid shared memory/message passing parallel machine. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I232–I326, 1993.

[Gal96] Mike Galles. The SGI SPIDER chip. Silicon Graphics Whitepaper, 1996.

[GBD$^+$94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

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

[GH93] Steven R. Goldschmidt and John L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–157, May 1993.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[GLR83] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for synchronizing large numbers of cooperating sequential processes. *ACM Transactions on Programming Languages*, 5(2):164–189, Apr 1983.

[GLS] William Gropp, Ewing Lusk, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Argonne National Labs Technical Report, `http://www.mcs.anl.gov/mpi/mpicharticle/-paper.html` and `mpi/mpicharticle.ps`.

[Gol93] Stephen Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance*. PhD thesis, Stanford University, June 1993.

[Goo97] James R. Goodman, June 1997. Personal communication.

[GT90] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, Jun 1990.

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

[HAA$^+$96] Mary M. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, Dec 1996.

[HBG$^+$97] John Heinlein, Robert P. Bosch, Jr., Kourosh Gharachorloo, Mendel Rosenblum, and Anoop Gupta. Coherent block data transfer in the FLASH multiprocessor. In *Proceedings of the Eleventh International Parallel Processing Symposium*, pages 18–27, April 1997.

[Hei] Mark Heinrich. *The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols*. PhD thesis, Stanford University. *to appear*.

[Hei97] Mark Heinrich, September 1997. Personal communication.

[Her90] Maurcice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles of Distributed Computing*, pages 197–206, March 1990.

[Her98] Stephen Alan Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998. Technical Report STAN-CS-TR-98-1603.

[HFM88] Debra Hengsen, Raphael Finkle, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, February 1988.

[HGDG94] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integrating message passing and shared memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[HGG94] John Heinlein, Kourosh Gharachorloo, and Anoop Gupta. Integrating multiple communication paradigms in high performance multiprocessors. Technical Report CSL-TR-94-604, Stanford University, Computer Systems Laboratory, February 1994.

[HHS$^+$95] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-94-660, Stanford University, Computer Systems Laboratory, 1995.

[HJ92] Dana S. Henry and Christopher F. Joerg. A tightly coupled processor-network interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, Sep 1992.

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

[HLS95] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, November 1995. A preliminary version appeared in the Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, July 1992.

[HM93a] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, May 1993.

[HM93b] Mark Homewood and Moray McLaren. Meiko CS-2 interconnect Elan-Elite design. In *Proceedings of Hot Interconnects 93*, August 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[HS95] Chris Holt and Jaswinder Pal Singh. Hierarchical n-body methods on shared address space multiprocessors. In *Proceedings of the Seventh SIAM International Conference on Parallel Processing for Scientific Computing*, pages 313–318, February 1995.

[HT93] W. Daniel Hillis and Lewis W. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, 36(11):30–40, November 1993.

[Int91] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[KA93] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the Alewife multiprocessor. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 195–206, July 1993.

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

[KCA92] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the window of vulnerability in multiphase memory transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–284, Oct 1992.

[KCD+97] Ravindra Kuramkote, John Carter, Alan Davis, Chen-Chi Kuo, Leigh Stoller, and Mark Swanson. Analysis of avalanche's shared memory architecture. Technical Report UUCS-97-008, University of Utah Computer Science Department, July 1997.

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

[KG98] Alain Kägi and James R. Goodman. SOFTQOLB: An ultra-efficient synchronization primitive for clusters of commodity workstations. Computer Sciences Department Tech Report 1327, University of Wisconsin, Madison, January 1998.

[KJA+93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message passing and shared-memory: Early experience. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 54–63, May 1993.

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

[Kus97] Jeff Kuskin. *The FLASH Multiprocessor: Designing a Flexible and Scalable System*. PhD thesis, Stanford University, November 1997. CSL-TR-97-744.

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

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

[Len92] Daniel E. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford University, February 1992.

[LL97] James Laudon and Daniel Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, Jun 1997.

[LLG$^+$90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

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

[Mar93] Margaret Martonosi. *Analyzing and Tuning Memory Performance in Sequential and Parallel Programs*. PhD thesis, Stanford University, Dec 1993.

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

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

[Mes93] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report No. CS-93-214, University of Tennessee, November 1993.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[Mic93] Microprocessor and Microcomputer Standards Subcommittee of the IEEE Computer Society, USA. *IEEE Standard for Scalable Coherent Interface (SCI),* August 1993. IEEE Std 1596-1992.

[MIP96] MIPS Computer Systems, Inc. MIPS R10000 microprocessor. `http://www.-mips.com/products/r10k`, 1996.

[MKAK94] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and M. Frans Kaashoek. FUGU: Implementing translation and protection in a multiuser, multimodel multiprocessor. Technical Report Technical Memo MIT/LCS/TM-503, MIT Laboratory for Computer Science, October 1994.

[MKF$^+$96] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Anant Agarwal, and M. Frans Kaashoek. UDM: User direct messaging for general-purpose multiprocessing. Technical report, MIT Laboratory for Computer Science, March 1996. Technical Memo MIT/LCS/TM-556.

[MLH94] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report SICS Research Report T94:07, Swedish Institute of Computer Science, Box 1263 S-164 28 Kista, Sweden, February 1994.

[MOH96] Margaret Martonosi, David Ofelt, and Mark Heinrich. Integrating performance monitoring and communication in parallel computers. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 138–147, May 1996.

[Mow94] Todd Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[NAB$^+$95] Andreas Nowatzyk, Gunes Aybay, Michael Browne, Edmund Kelly, Michael Parkin, Bill Radke, and Sanjay Vishin. The S3.mp scalable shared memory multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, August 1995. vol 1 of 3.

[Now97] Andreas Nowatzyk, July 1997. Personal communication.

[NPA92] Rishiyur Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156–167, May 1992.

[NWD93] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224–235, May 1993.

[PBG$^+$85] G. F Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Processor Prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, Aug 1985.

[Pie88] Paul Pierce. The NX/2 operating system. In G. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume 1 of 2, pages 384–390, 1988.

[RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer simulation: The SimOS approach. In *IEEE Parallel and Distributed Technology*, pages 34–43, 1995. Winter.

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

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

[SCH$^+$91] Chriss Stephens, Bryce Cogswell, John Heinlein, Greg Palmer, and John Shen. Instruction-level profiling and evaluation of the IBM RS/6000. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 180–189, May 1991.

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

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

[SH98] Ioannis Schoinas and Mark D. Hill. Fine-grain access control for distributed shared memory. In *Proceedings of the Fourth International Symposium on High Performance Computer Architecture*, February 1998.

[Sim92] Richard Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. PhD thesis, Stanford University, October 1992.

[Smi92] Michael David Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, November 1992.

[SPG91] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Principles*. Addison-Wesley, third edition, 1991.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[ST95] Nir Shavit and Dan Toutitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[Sta93] Richard Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, MA, June 1993.

[Sto90] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, 1990.

[Sun95] Sun Microsystems, Inc. The UltraSPARC processor — technology white paper. `http://www.sun.com/sparc/whitepapers/UltraSPARCtechnology`, 1995.

[SW95] Richard L. Sites and Richard T. Witek, editors. *Alpha AXP Architecture Reference Manual*. Digital Press, 1995. Second Edition.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992. Also available as Stanford University Computer Systems Laboratory Tech Report CSL-TR-91-469.

[TBG$^+$97] Dan Teodosiu, Joel Baxter, Kinshuk Govil, John Chapin, Mendel Rosenblum, and Mark Horowitz. Hardware fault containment in scalable shared-memory multiprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997. To appear.

[Thi91] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

[Thi92] Thinking Machines Corporation. *Programming the NI*, Mar 1992.

[VDGR96] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

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

[Vee86] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

[WG89] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[WG91] Philip J. Woest and James R. Goodman. An analysis of synchronization mechanisms in shared-memory multiprocessors. Computer Sciences Department Tech Report 1005, University of Wisconsin, Madison, Feb 1991. Also appeared in *Proceedings of the 1991 International Symposium on Shared Memory Multiprocessing*.

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

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–216, Dec 1993.

[Woo96] Steven Cameron Woo. *The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors*. PhD thesis, Stanford University, May 1996.

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

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John Hennessy. The performance advantages of integrating block transfer in cache coherent multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.