THE FLASH MULTIPROCESSOR: DESIGNING A FLEXIBLE AND SCALABLE SYSTEM

Jeffrey Scott Kuskin

Technical Report No. CSL-TR-97-744

November 1997

This research has been supported by DARPA contract DABT63-94-C-0054.

The FLASH Multiprocessor: Designing a Flexible and Scalable System

Jeffrey Scott Kuskin

CSL-TR-97-744

November 1997

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
William Gates Computer Science Building, A-408
Stanford, CA 94305-9040
pubs@shasta.stanford.edu

Abstract

The choice of a communication paradigm, or protocol, is central to the design of a large-scale multiprocessor system. Unlike traditional multiprocessors, the FLASH machine uses a programmable node controller, called MAGIC, to implement all protocol processing. The architecture of the MAGIC chip allows FLASH to support multiple communication paradigms — in particular, cache-coherent shared memory and high-performance message passing — while minimizing both hardware and software overhead. Each node in FLASH contains a microprocessor, a portion of the machine's global memory, a port to the interconnection network, an I/O interface, and MAGIC, the custom node controller. The MAGIC chip handles all communication both within the node and among nodes, using hardwired data paths for efficient data movement and a programmable processor optimized for executing protocol operations. The result is a system that is flexible and scalable, yet competitive in performance with a traditional multiprocessor that implements a single communication paradigm completely in hardware.

The focus of this dissertation is the architecture, design, and performance of FLASH. Much of the motivation behind the FLASH system and the MAGIC node controller design stems from an examination of the characteristics of protocol code and the architecture of the DASH system, the predecessor to FLASH. This examination led to two major design goals: development of a node controller architecture that can attain high protocol processing performance while still maintaining flexibility and a need to reduce the logic and memory overheads associated with cache coherence. The MAGIC design achieves these goals by implementing on a single chip a programmable protocol engine with an instruction set optimized for the characteristics of protocol code, along with dedicated support

logic to alleviate the most serious protocol processing performance bottlenecks — data movement, message dispatch, and lack of close coupling to the node board components. The design of the FLASH node complements the MAGIC design, matching the close coupling and high bandwidth support in MAGIC to provide a balanced node architecture.

Next, the dissertation investigates the performance of cache-coherence on FLASH. Performance results are presented from microbenchmarks run on the Verilog RTL of the MAGIC chip and from complete applications run on FlashLite, the FLASH system-level simulator. The microbenchmarks demonstrate that the architectural extensions added to the MAGIC design — particularly the instruction set optimizations to the programmable protocol processor — yield significantly lower latencies and protocol processor occupancies to service the most common types of memory operations.

The application results are used to evaluate the performance costs of flexibility by comparing the performance of FLASH to that of a hardwired machine on representative parallel applications and multiprogramming workloads. These results show that poor application memory reference or load balancing characteristics cause the performance of the FLASH system to degrade more rapidly than the performance of the hardwired system; that is, FLASH's performance is less robust. For applications that incur a large number of remote misses or exhibit substantial hot-spotting, the increased remote access latencies or the occupancy of MAGIC lead to lower performance for the flexible design.

Overall, however, the performance of FLASH can be competitive with the performance of the hardwired machine. Specifically, for a range of optimized parallel applications, the performance differences between the hardwired machine and FLASH are small, typically less than 10% at 32 processors and less than 15% at 64 processors. For these programs, either the processor cache miss rates are small or the latency of the programmable protocol processing can be hidden behind the memory access time.

Keywords and Phrases: distributed shared memory, cache coherence, message passing, scalable multiprocessors, communication protocols, parallel processing.

Acknowledgments

The FLASH design was a huge effort, especially in a university setting. A number of people were instrumental in carrying the design from concept to actual hardware. First, I would like to thank John Hennessy, my primary advisor. John's inspiration and confidence have been unceasing during my tenure at Stanford and his thorough review of this dissertation has greatly improved its quality. I also would like to thank Mark Horowitz, both for acting as my secondary advisor and for serving as the day-to-day manager of the FLASH hardware development effort. In addition, I thank John Cioffi for graciously agreeing to chair my Ph.D. orals committee and for serving on my dissertation reading committee.

I owe a huge debt of gratitude to the members of the FLASH hardware team, the group responsible for the design and implementation of the MAGIC chip and the FLASH system. Their dedication, talent, and sheer perseverance were extraordinary:

- Joel Baxter — Created much of the protocol code development environment and support tools.
- Jules Bergmann — Designed and implemented the protocol processor, the most complex unit on the MAGIC chip.
- John Heinlein — Developed the message passing protocol code.
- Mark Heinrich — Key developer of FlashLite and several cache coherence protocols. Designed and implemented the inbox.
- Hema Kapadia — Architect of the IO unit.
- Dave Nakahira — Designed and implemented the memory controller, the JTAG unit, and the diagnostic port. Single-handedly designed the FLASH node board.
- Dave Ofelt — Designed and implemented the PI and much of the HLL environment. Managed the diagnostic regression suite. Co-old-timer on the FLASH team.
- Jeff Solomon — MAGIC place and route, hardmac development, and other physical design tasks.
- Ron Ho and Evelina Yeung — Designed the data buffer memory array.

Many other students aided in the design and verification as well, including Jisu Bhattacharya, Luke Chang, Jeff Gibson, Ziyad Hakura, Richard Ho, and Rich Simoni. The MAGIC design also benefited from the contributions of the FLASH operating system group, under the leadership of Mendel Rosenblum. In addition, I would like to acknowledge the assistance of Mike Galles, Jim Laudon, and Dan Lenoski at Silicon Graphics. Mike Galles, in particular, supplied the LLP logic and cheerfully answered many questions from me about its operation and use.

A special thanks to my fellow research group members and (former) trailerites — Steve Woo, Chris Holt, and Andrew Erlichson — and to the members of the FLASH softball team.

I would like to thank my parents for their love, support, and encouragement throughout my Ph.D. career.

Most important, I would like to thank my wife, Diane. Being married to a seemingly perpetual student surely was not easy, and Diane's unfailing love, encouragement, and sense of humor have been constant reassurances on the long road to a degree.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Within the last several years, multiprocessors have gained wide acceptance in the computer market. What were once esoteric machines reserved mostly for scientific computation are now employed routinely in general-purpose commercial, technical, and computer-aided design settings. Even manufacturers of low-end business and personal computers offer multiprocessor systems, heralding the movement of these systems into the highest volume market segment.

Despite their rising popularity, the vast majority of multiprocessors in use are small-scale systems containing just a few — perhaps four or eight — off-the-shelf microprocessors. Such systems are relatively easy to build, make cost-effective use of commercial microprocessor and system packaging technologies, and deliver significantly increased performance for many applications. Although small-scale multiprocessors work well, their aggregate computational power is limited necessarily by the small number of processors. Many problems require much greater computational power than a small-scale system can provide, leading to an interest in constructing larger systems containing hundreds or even thousands of processors.

Like their small-scale counterparts, most large-scale multiprocessors employ off-the-shelf microprocessors to leverage the tremendous computational capability and cost/performance available. Beyond the use of commercial microprocessors, however, large-scale multiprocessor design differs substantially from the design of small-scale machines. Large-scale systems bring a variety of architectural, design, and implementation challenges, many of which are related to the way the processors communicate with each other and with memory. For example, a typical small-scale system achieves communication by connecting all processors and memory modules to a common bus whereas large-scale systems use a much greater variety of communication mechanisms.

As Figure 1.1 illustrates, the overall multiprocessor design space can be organized as a hierarchy of communication mechanisms. Small-scale machines belong to the centralized memory portion of the hierarchy, often called uniform memory access (UMA) designs. As noted above, these systems connect all processors and memory modules to a common bus, resulting in a single, centralized memory that is equally accessible to all CPUs. For large-scale machines, scalability limitations preclude the use of a common bus. Instead, both the processors and memory modules are distributed throughout the system. Because memory

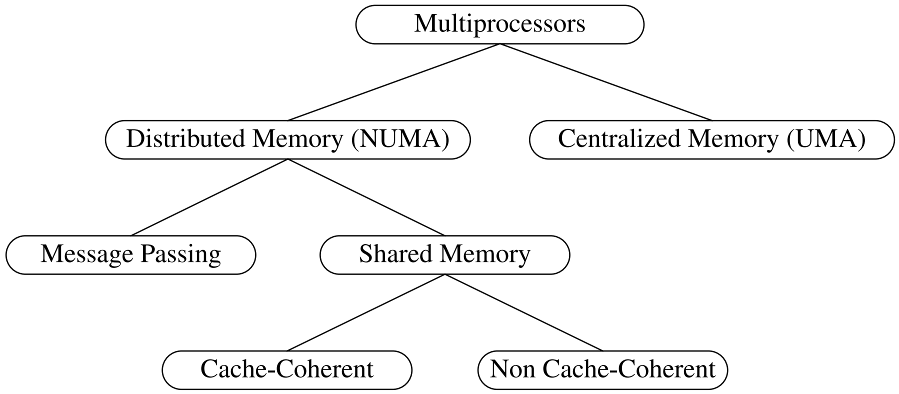


Figure 1.1 Multiprocessor design space

is distributed, a particular processor can access some memory locations more quickly than others, leading to a non-uniform memory access (NUMA) design.

Within the distributed memory framework, a number of additional communication options help to manage the potential performance loss caused by the system's NUMA characteristics. In a message passing machine, processors can communicate directly only with local memory. Communication with other processors or more distant memories must be requested explicitly in the application. Shared memory machines, in contrast, make all communication implicit. When a processor requests a memory location, the system fetches the needed value regardless of how long the access will take or which memory contains the data. Implicit communication also requires shared memory systems to cope with the cache coherence issue — whether processors are permitted to cache data after it has been retrieved from the memory system. Although caching has obvious performance advantages, implementing cache coherence is complex; hence the development of both cache-coherent and non cache-coherent systems.

1.1 The FLASH Project

Initially, design and development of large-scale, distributed multiprocessors proceeded mostly in the research community. A number of systems were constructed [2, 33, 36, 65] and many of the tradeoffs inherent in the design options described above were explored, both through study of actual systems and by simulation. The success of these research efforts led to commercial acceptance and development of distributed multiprocessors [7, 16, 59, 64, 68, 94, 107]. Although some of the commercial systems support large processor counts, the focus of the commercial multiprocessor vendors is primarily on relatively modest system sizes.

Considerable interest remained in developing systems with hundreds or thousands of processors. Yet, little is known about how to design systems of this size — tradeoffs appropriate for a few tens of processors might be undesirable for larger processor counts. The FLASH effort addresses this problem by providing a means to quantitatively evaluate design options for next-generation, large-scale distributed multiprocessors. Rather than concentrating on the smaller distributed systems that have been investigated thoroughly in the research community and are being developed actively by commercial vendors, FLASH concentrates instead on extending distributed multiprocessor design to explore the challenges of future architectures.

Many of these challenges concern scalability — that is, ensuring that the overall performance of a system continues to increase proportionally as more processors are added. Achieving a scalable system is difficult even at fairly small processor counts but is necessary to warrant the additional cost of a multiprocessor. With hundreds or thousands of processors, achieving a scalable system becomes even more formidable.

The FLASH system is designed to allow investigation of scalability and several other research topics:

- What communication mechanisms should be provided? Message passing and shared memory each have proponents and the facilities to implement these mechanisms are similar. Can a single machine provide effective support for both shared memory and message passing?
- How should communication protocols be designed to function efficiently on large-scale machines? Implementation of cache-coherent shared memory is particularly demanding as system size grows.
- What performance statistics and data collection facilities are desirable to evaluate application, operating system, and communication protocol performance?
- What hardware support — such as for synchronization — is needed to allow applications, as well as the operating system itself, to achieve good scalability and good overall performance at large processor counts?

The FLASH project addresses these issues by designing and implementing a multiprocessor system based on a programmable protocol engine, thereby providing support for many different communication protocols and allowing easy and flexible performance data collection. By providing this framework, the FLASH system — the acronym stands for FLexible Architecture for SHared memory — permits quantitative research on an actual machine into next-generation multiprocessor design issues. Without machines such as FLASH, these issues could be investigated only on a simulator (if at all), and often with significant compromises to allow the simulation to complete in a reasonable amount of time.

Design and development of a real machine capable of efficiently supporting a variety of communication protocols is central to the FLASH goals outlined above, and is the subject of this dissertation.

1.2 Research Contributions

The primary contributions of this dissertation are:

- Investigation of the architecture and implementation issues involved in the design of a large-scale multiprocessor making use of a flexible, programmable protocol engine. Achieving a system that is both flexible enough to support a variety of communication protocols and that offers performance competitive with a machine designed specifically for one communication protocol is the goal of this investigation.
- The architecture and design of the MAGIC chip, a 750K-gate ASIC containing a programmable protocol engine, memory controller, network interface, processor interface, PCI controller, and significant additional support logic to ensure good system performance across a range of communication protocols.
- Analysis of the performance of the FLASH system on actual parallel scientific applications and multiprogrammed workloads. This analysis, based on a detailed simulation of the FLASH system, demonstrates that FLASH is scalable — it can achieve good speedups across a range of processor counts on a variety of applications — and that FLASH can achieve performance comparable to a system optimized for a single communication protocol.

1.3 Organization of the Dissertation

The beginning of this chapter presented a brief overview of the multiprocessor design space. Chapter 2 explores multiprocessor design in more detail, including a discussion of the cache coherence problem that is central to distributed shared memory systems. In addition, it describes the DASH system. DASH, also developed at Stanford, is the predecessor to FLASH and serves as the starting point for much of the FLASH research.

Chapter 3 describes FLASH in more detail, focusing specifically on the motivation behind and evolution of the MAGIC chip, FLASH's flexible protocol engine. The chapter also presents an overview of the FLASH node board, system packaging, and interconnection network.

Chapter 4 describes the internal architecture of MAGIC in greater detail, beginning with an overview of the chip's architectural features and continuing with a detailed discussion of the functional units. The chapter concludes with a discussion of how MAGIC was implemented.

Chapter 5 focuses on how MAGIC is used. The chapter describes two cache coherence protocols developed for use on FLASH and provides a demonstration of how the architectural goal of supporting multiple communication protocols via MAGIC's flexible protocol engine is achieved in practice.

Along with supporting multiple communication protocols, an equally important goal of the FLASH design is to achieve performance competitive with a machine optimized for a single communication protocol. Chapter 6 investigates FLASH system performance using microbenchmarks as well as complete parallel scientific applications and a collection of multiprogrammed workloads.

Chapter 7 contrasts a number of system designs, from both commercial vendors and academia, to FLASH. These systems span a range of architectural styles and design goals.

Concluding the dissertation, Chapter 8 summarizes the major results of this research and discusses some potential areas for future work.

Chapter 2

Background

This chapter first presents an introduction to distributed multiprocessors and some of the associated design challenges, including how to achieve a scalable system and the mechanisms for communicating among the processing nodes. Next, the chapter describes the DASH system, the predecessor to FLASH. The description outlines the basic system architecture of DASH and identifies a number of limitations resulting from its design and implementation. These limitations serve as motivation for some aspects of the FLASH system design, to be presented in Chapter 3.

2.1 Distributed Multiprocessing

Traditionally, small-scale multiprocessors have been implemented using a “snoopy” bus architecture, illustrated in Figure 2.1, in which all processors and memory modules are connected to a common, shared bus, thereby allowing a memory request issued by one processor to be observed by all other processors and memories. Each processor compares the address of the memory request to the contents of its cache and maintains cache coherence by combining the results of these comparisons to determine whether the latest copy of the requested data resides in memory or in one of the processor caches. Using a shared bus simplifies the implementation of cache coherence both by allowing the processors and memories to communicate and by providing the broadcast mechanism necessary to implement snooping. Unfortunately, the use of a bus also makes this style of multiprocessor non-scalable. As more and more processors are added to the system, the difficulties of proportionally increasing bus bandwidth while maintaining low snoop response times severely limit any increase in overall system performance.

To overcome the scalability limitations of bus-based multiprocessors, scalable multiprocessors replace the shared bus with a point-to-point interconnection network, resulting in a distributed system containing a collection of processing nodes connected by a point-to-point communication network, as shown in Figure 2.2. Each processing node contains a compute processor with its cache, a portion of the machine’s distributed memory, and a node controller (NC). The node controller is responsible for managing all communication both among the components on a single processing node as well as communication between two processing nodes via the interconnection network.

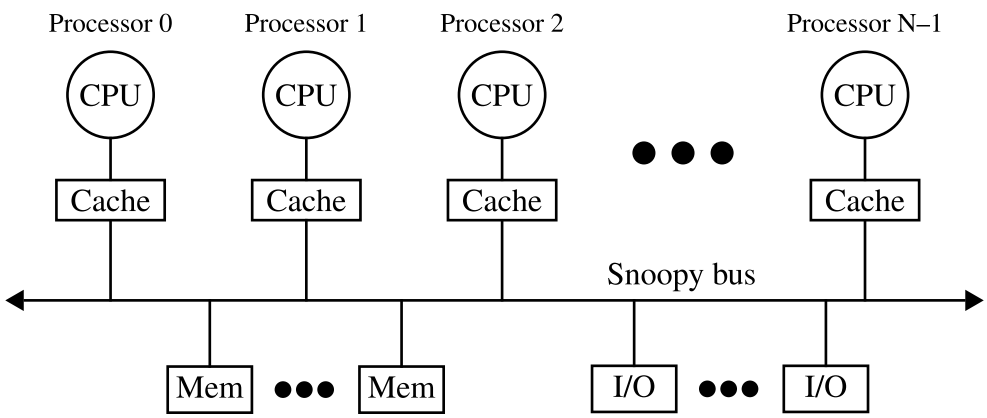


Figure 2.1 Bus-based multiprocessor system

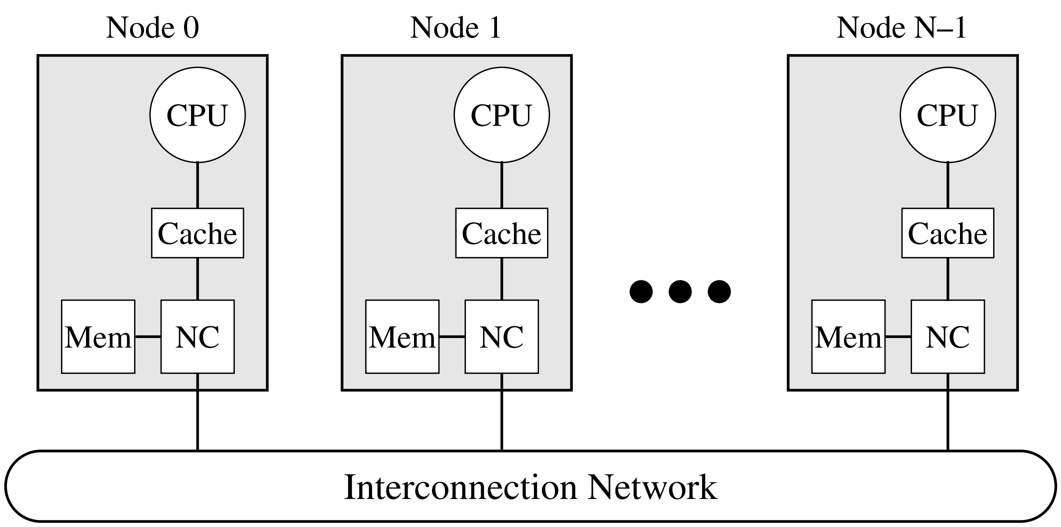


Figure 2.2 Distributed multiprocessor system

Unlike in a bus-based system, memory requests issued by a compute processor in a distributed system are visible only to the local node controller rather than to all processors and memories. To achieve a usable multiprocessor system, then, the processing nodes must communicate; without communication, the system would appear as just a collection of isolated processing nodes with no ability to utilize memory, computation capability, or other resources from the other processing nodes.

2.1.1 Communication Paradigms

The convention the processing nodes use to communicate is called a *communication paradigm*. The long latency required to communicate among nodes in a distributed multiprocessor makes the choice of communication paradigm critical to overall system performance. A paradigm must provide mechanisms to manage and tolerate this latency as well as the means to achieve high global data bandwidth. In addition, the paradigm needs to export a useful interface to the programmer so that parallel applications can be developed to fully exploit the capabilities of the system.

Two major communication paradigms are in use today: message passing and shared memory. In a *message passing* system, each compute processor has direct access only to its local memory, that is, only to the memory stored on its local processing node. Communication between nodes must be requested explicitly in a message passing system, typically by augmenting application code with calls to a library of communication routines that permit user-specified data to be exchanged among the processing nodes' memories.

In a *shared memory* system, all of the physically distributed memories on the processing nodes are made to appear as a single, unified address space. Each compute processor can access any memory location in the system via normal load and store instructions and the node controllers communicate to transfer the necessary data when a processor requests data not stored in its node's local memory. In this way, communication among processing nodes in a shared memory system is implicit and occurs whenever a compute processor references a remote address, thereby preserving the illusion that all of system memory is equally accessible to all processors.

2.1.2 Message Passing versus Shared Memory

Message passing systems as well as distributed shared memory systems have been built, both in academia and commercially [3, 47, 53, 65, 80, 94]. A natural question is whether one communication paradigm is superior to the other. This question is difficult to answer definitively since each paradigm has advantages and drawbacks, depending on the size of the machine and the nature of the applications it is running.

2.1.2.1 Message Passing

Typically, message passing machines are regarded as more easily scalable. Systems containing hundreds of processors are constructed routinely [47, 53], and a message-passing system with several thousand processors became operational recently at Sandia National Laboratory [54]. The explicit nature of inter-node communication tends to ease construction of large message-passing systems. In addition, providing the programmer with explicit control over inter-node communication is sometimes regarded as an advantage in constructing large-scale programs to run on these systems.

Explicit communication also has the advantage of combining data transfer with synchronization — when a message is received from another node, the data it contains is by definition ready for consumption by the receiving processor and is not subject to further modification by the sender. A separate mechanism for determining when the data may be used safely is not needed. In addition, explicit requests for data transfer decouple the actual data transfer process from the computation needed to formulate the data to be sent. As a result, data transfer and communication often can be overlapped, meaning that the node controller accepts responsibility for data transfer, thereby improving performance by allowing the compute processor to proceed with further computation while the data transfer occurs.

The major drawback of the message passing paradigm is its inefficiency with unstructured algorithms and fine-grained communication. By making communication explicit, the cost of invoking inter-node communication is higher. Best performance typically is achieved only if the amount of data to be transferred is large, the need to perform the transfer is known well before the recipient requires the data, and the data to be sent can be identified early. Usually, none of these three requirements is met in an unstructured or fine-grained code. For instance, a tree-based algorithm such as the Barnes-Hut N-body code [48] communicates individual tree node data structures among the processors. These tree nodes are typically only a hundred bytes in size or so, and thus incur significant data transfer overhead. Worse, the exact tree nodes that need to be transferred can be identified only just before their contents are needed for computation, greatly reducing overall application performance by effectively serializing communication and computation rather than allowing the two to be overlapped.

2.1.2.2 Shared Memory

Shared memory, in contrast, supports fine-grained, unstructured communication very efficiently. Because communication is invoked implicitly when a processor requests remote data, data transfer is handled directly by the system hardware, minimizing overhead. More important, the shared memory paradigm integrates communication directly into the system programming model, eliminating the need to explicitly request and manage communication and easing the programming effort both in terms of algorithm development and in terms of implementation by the programmer and compiler.

Furthermore, the implicit communication of the shared memory paradigm naturally admits caching into the system implementation. Modern processors depend on local, closely-connected caches to improve performance. From a more global perspective, caches are crucial for maximizing inter-node communication bandwidth and for minimizing the latency to access remote data by exploiting the spacial and temporal locality present in both local and remote memory references. Caches also permit data prefetching [75], another critical mechanism for reducing the effective remote memory access latency.

2.1.2.3 Cache-Coherent Shared Memory

Because of the critical role caches play in maximizing inter-node communication bandwidth, minimizing remote access latency, and improving overall system performance, most shared memory systems extend the shared memory paradigm to provide what is known as cache-coherent shared memory. In a cache-coherent system, not only are all the physically distributed memories made to appear as a single, unified address space, but each processor's view of this memory is kept consistent. In other words, these systems recognize that when a processor requests a memory location, the corresponding data will be stored in that processor's cache, thereby allowing subsequent accesses to retrieve the data from the processor's cache rather than from a remote memory.

Caching complicates the shared memory paradigm in two ways. First, if a processor updates (writes to) a memory location in its cache, then the most up-to-date version of that memory location is in the processor's cache and not in memory. Subsequent requests for the memory location must be retrieved from the cache and not from memory. Second, several processors can cache the same memory location simultaneously. Thus when a processor updates a memory location, all other copies of that location must be removed from the processor caches or else a processor could continue to read the old value of the memory location from its cache rather than the updated value. Figure 2.3 illustrates the cache-coherence problem, showing in the top half of the figure two processors (P0 and P2) reading and caching the same memory location X. When P0 updates X, as shown in the bottom half of the figure, the new value is stored in P0's cache and, potentially, propagated to memory as well. The old value of X remains in P2's cache, however, and subsequent reads of X by P2 will retrieve the old value from its cache rather than the updated value. Cache-coherent systems must solve this problem and ensure, in this example, that P2 is notified when P0 updates location X.

The need to determine where the most up-to-date copy of a cache line is stored is common to both bus-based and distributed systems. Indeed, tracking the location of cache lines efficiently and quickly as the system size grows is a significant challenge and is the root of the most prominent criticism of distributed cache-coherent systems: that they are not easily scaled, particularly not to the processor counts achieved with message passing

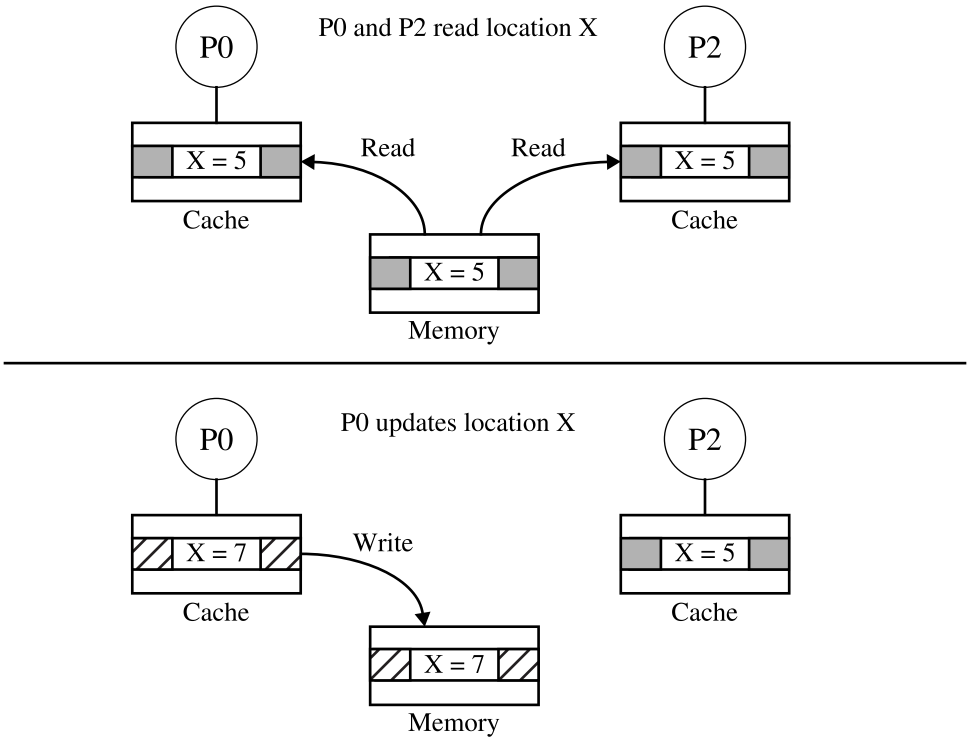


Figure 2.3 Cache coherence problem

systems. However, if these scalability concerns can be addressed then an important advantage of distributed cache-coherent shared memory can be realized: it preserves the same system programming model provided on small-scale, bus-based multiprocessors. As a result, the same familiar development and execution environment is available on large-scale systems, even though the underlying system architecture is very different. Programmers can develop code that will run unchanged on both bus-based and distributed systems, and users need not be concerned with managing different binaries for the two systems. Of course optimal performance on large-scale distributed systems often demands algorithmic modifications to adapt to the non-uniform local and remote memory access latencies. However, these changes are optimizations rather than requirements.

2.1.2.4 Integrated Message Passing and Cache-Coherent Shared Memory

Message passing and cache-coherent shared memory each have advantages and disadvantages both at the system level and for the programmer. Most systems choose just one of these paradigms, forcing the programmer to accommodate the particular paradigm the system provides. A more attractive option is to provide integrated support for both message passing and cache-coherent shared memory in a single system, allowing the programmer to reap the benefits of each. The challenge at a system level is integrating the two paradigms so that both message passing operations and cache coherence can be used simultaneously without requiring the programmer to designate separate memory regions for message passing and cache coherence. This integration is key if a combination of the two paradigms is to be used in a single application, thereby allowing the bulk data transport and communication-computation overlap advantages of message passing to be combined with the fine-grained communication efficiency of cache coherence.

Particularly as system size grows, the tradeoffs between cache-coherent shared memory and message passing are unclear. Providing both paradigms allows exploration into how to combine the two in a single application and whether overall application performance increases as a result. One of the primary goals of FLASH, as will be described in Section 3.1, is to provide efficient, integrated support for both cache-coherent shared memory and message passing while not compromising the individual performance of either paradigm.

2.1.3 Distributed Cache Coherence

Use of a point-to-point interconnection network poses particular challenges for implementing cache-coherent shared memory. Although connecting the processing nodes in this manner yields a scalable system, it also eliminates the natural broadcast capabilities of a shared bus that the snoopy bus-based system requires to implement cache coherence. Instead, scalable systems must use a distributed cache coherence protocol that relies on explicit node-to-node messages rather than system-wide broadcast to maintain cache coherence. Fundamentally, these systems must track sharing information; that is, which memory locations each processor is caching. Sharing information is kept in an auxiliary data structure called a *directory*, illustrated in Figure 2.4.

Like the system's memory, the directory information is distributed among the processing nodes to avoid the performance bottleneck of a single, monolithic directory. Each node maintains a directory corresponding to the locations in that node's local memory; Figure 2.4 shows an example of one node's directory contents. The directory consists of a collection of directory entries, one for each memory block in the node's local memory. Because the processor caches interface to the system at a cache line granularity — that is, each processor cache miss or writeback transfers a single cache line of data between memory and the cache — the size of the memory block tracked by each directory entry is usually one cache line. In its simplest form, a directory entry contains two fields: a state indication and a presence bit vector. The state indication specifies whether the memory line associated with the directory entry is held shared (i.e., read-only) in one or more

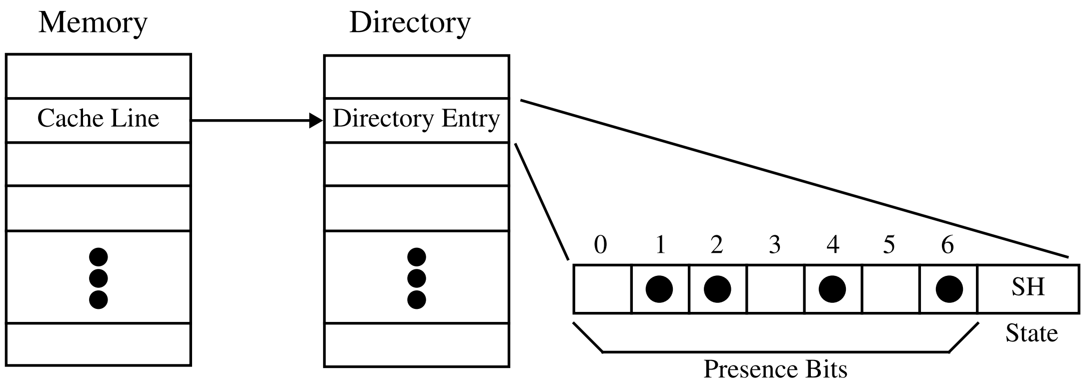


Figure 2.4 Example directory implementation

caches or whether it is held exclusive (i.e., with read/write permission) in a single processor's cache. In both cases the presence bit vector indicates which processors are caching the memory line; if the memory line is held exclusive, only one presence bit may be set. The directory entry in Figure 2.4 shows a case in which the corresponding memory line is held shared, indicated symbolically by the "SH" in the state field, in the caches of processors 1, 2, 4, and 6, indicated by the presence bit vector.

When a memory request arrives at a processing node, the node controller retrieves the corresponding directory entry to determine what additional actions are required to service the request. In the case of Figure 2.4, for example, if processor 3 requested exclusive access to the memory line, the memory line first must be removed, or invalidated, from all processor caches currently holding it. In a bus-based system, these invalidations would be performed automatically when processor 3's exclusive request was issued on the snoopy bus. In a distributed system, however, the node controller must consult the presence bit vector to determine that explicit invalidation messages need to be sent to processors 1, 2, 4, and 6. This case is just one example of the operation of a distributed cache coherence protocol. In practice, these protocols are complex, especially because so many race conditions can occur as a result of the lack of a shared bus to serialize all processors' memory requests. More details of distributed cache coherence protocol implementation can be found in [46, 65].

2.2 The DASH System

Section 2.1 motivated the need for a distributed multiprocessor system architecture to overcome the scalability limitations of the traditional bus-based design. Like their bus-based counterparts, distributed multiprocessors must provide a cache-coherent shared memory programming model, yet the lack of broadcast and snoop capability in a distributed system significantly increases the complexity of the cache coherence implementation. The goal of the Stanford DASH system [65, 66, 67], the predecessor to FLASH, was to demonstrate that, despite these complexities, a cache-coherent distributed shared memory (CC-NUMA) system could be constructed. Although the foundations of CC-NUMA multiprocessor design — such as the use of a directory to track caching information and the basic operation of a distributed cache-coherence protocol — were developed before the DASH design started [18, 114], no actual CC-NUMA systems had been built. Of particular concern was whether a CC-NUMA system could be implemented at a reasonable total cost, and whether good overall system performance and cost-performance could be achieved as the system size scaled. The primary goal of DASH was to construct a prototype system to demonstrate that a cost-competitive, scalable CC-NUMA system could be built. As a consequence of this goal, the DASH system employed a simple, straightforward directory structure and implemented the entire cache-coherence protocol and all directory processing completely in hardware. A 16-processor DASH system became operational in early 1992.

A number of researchers used the DASH system to investigate application and operating system performance on a CC-NUMA system [72, 104]. The existence of DASH made studies such as these, using an actual machine, operating system, and realistic problem sizes, possible. At the same time, these studies exposed a limitation of the DASH design: inflexibility. In other words, the DASH system implemented one particular directory structure and a single cache-coherence protocol, all in a set of hardwired state machines and datapaths within the node controller that could not be changed. The comment that DASH is inflexible is not a criticism of its design; rather, the decisions to use a simple directory structure and cache-coherence protocol and to implement all protocol processing in a hardwired fashion were key factors in achieving a design that was buildable and could demonstrate the feasibility of CC-NUMA multiprocessors. A more complex design was not needed and in fact would have been undesirable. However, to a much greater extent than on a bus-based multiprocessor, performance on a CC-NUMA system is a function not only of the underlying application algorithms and data structures, but also of the interactions between the algorithms and data structures and the system's cache coherence implementation. The inability to alter DASH's directory organization and cache coherence protocol made study of the interaction of the system, operating system, and applications very difficult.

The DASH prototype successfully demonstrated the feasibility and scalability of a CC-NUMA system. While the success of DASH encouraged development of commercial CC-NUMA systems, DASH's success also meant that further research efforts with a similar proof-of-concept goal as DASH were unnecessary. At the same time, both the success of DASH and the inflexibility of its design motivated the exploration of a new, flexible multiprocessor architecture that took the feasibility of CC-NUMA as a given and instead focused on supporting the exploration of other aspects of large-scale multiprocessor design. The FLASH machine is designed to achieve this goal.

The major limitation of the DASH design, as noted above, was the use of a single, hardwired cache coherence protocol. DASH also supported only a limited set of predefined synchronization primitives, all completely hardwired, some of which were used extensively and some not used at all [65]. In addition, DASH provided only limited performance monitoring facilities. These facilities were implemented mostly in a field-programmable gate array (FPGA), making them hard to modify without knowledge of hardware design and FPGA tools. The FPGA could monitor only a predefined set of inputs, thus limiting the set of possible performance measurements, and all performance data was stored in a dedicated memory, limiting the amount and duration of data that could be collected. Furthermore, even though DASH achieved good performance and good cost-performance, the logic and memory overhead of adding cache coherence to the system was significant — about 20% [65] — and added measurably to the total system cost. Finally, DASH supported only the cache-coherent shared memory communication paradigm; it provided no support for message passing.

FLASH seeks to build on the success of DASH by overcoming the limitations identified in DASH, by extending the range of system sizes that perform well using cache-coherent shared memory, and by allowing much more flexible and comprehensive research into large-scale, distributed multiprocessor design. The next chapter describes the specifics of the FLASH design goals in more detail.

Chapter 3

FLASH System Architecture

This chapter discusses the rationale behind the FLASH system architecture and the internal architecture of the MAGIC component. It first describes a number of high-level goals the FLASH system is designed to address. Next it presents the FLASH node architecture, followed by a description of a series of performance studies that helped shape the MAGIC architecture.

3.1 Motivation and Goals

In addition to identifying the limitations of DASH as described in Section 2.2, the FLASH architecture was guided by the need to explore and quantitatively evaluate the design issues involved in constructing large-scale multiprocessors. Four major research issues were identified: communication paradigms, cache coherence implementation, scalability, and reliability.

First, what communication paradigms, or programming models, should a machine support? Cache-coherent shared memory and message passing are the two dominant communication paradigms in use, and research focused on the relative performance and ease of implementation of the two paradigms, as well as on their scalability and whether integrated support for both paradigms in a single system is desirable.

Second, how should cache coherence be implemented? Debate centered over the relative merits of a cache-only memory architecture (COMA) versus a more traditional CC-NUMA approach [55, 97], but also included investigation into variations on the basic write-invalidate protocol used in DASH, variations on directory data structure formats [1, 19, 22, 95, 108, 112], and the desirable set of synchronization primitives needed for both small- and large-scale systems.

Third, scalability research focused on how the performance of applications, the operating system, and the communication protocols scale as the machine size grows. The goal of FLASH in this area is to design a system that can scale, potentially, to a total processor count on the order of 1000. Little is known about the performance and scalability characteristics of large-scale machines running large parallel applications. In addition, even on existing smaller-scale machines, performance data applicable to large-scale machines is difficult to collect because these machines are not designed to scale to large numbers of

CPUs, often contain almost no performance monitoring facilities (DASH was a notable exception), and provide few if any tools to analyze whatever performance data can be collected.

Fourth, research focused on improving the reliability of large-scale systems. Distributed multiprocessors, in particular those providing a shared memory communication paradigm, suffer from a lack of reliability. Failure of a single processor or of the operating system itself usually necessitates a complete system reboot, even though most of the system is still operational. The most common defense against this problem is periodic checkpointing within the application followed by a restart from the checkpoint after the system has crashed and been rebooted. A more attractive solution is to eliminate the need to reboot after a single component failure, and the hardware and protocol features needed to support this goal were at the center of this research question.

3.1.1 FLASH System Features

With these four issues and the DASH limitations in mind, the next step was to determine what machine features were needed to support large-scale multiprocessor research. Clearly the system had to support multiple communication paradigms and the protocols to implement them. Such support enables the evaluation of the tradeoffs of cache-coherent shared memory versus message passing as well as protocol variation studies. Hence, support for both cache-coherent shared memory and message passing was required, and ideally the system would provide this support in an integrated fashion so that the two communication paradigms could be used simultaneously, potentially by a single application.

In addition to supporting multiple protocols, the system also needed to support a range of performance monitoring features. Because a multiprocessor operating environment is complex, the ability to gather accurate data about system performance is critical to understanding application behavior and for tuning applications and protocols to maximize overall performance. The system performance monitoring facilities must be comprehensive enough to allow results to be quantified to varying, and potentially very fine, levels of detail and they must allow almost arbitrary performance data to be collected so as to satisfy the often incompatible needs of architecture and protocol research (typically very low-level data), operating system research (mid-level data), and application research (low-, mid-, and high-level data). A performance monitoring implementation that permitted only a single, predefined set of performance data to be collected would have been inadequate. Flexibility to collect a variety of performance data was needed in the same way that flexibility was needed to support multiple protocols.

Most important, the system needed to support multiple protocols and performance monitoring while maintaining good performance across all feature variations. The exact meaning of the term “good” in the previous sentence is crucial. For FLASH, good performance is defined as performance that is competitive with a machine that retains the basic FLASH system architecture but implements a single protocol completely in hardware. If the DASH design were updated to the same technology as FLASH, for example, it would satisfy these criteria and its performance could be used as a reference for a FLASH system configured to use the same protocol and directory organization. Achieving competitive performance is perhaps the biggest design challenge, yet it is critical if the system performance results are to be believable and applicable to other architectures. Designing a system that can meet the design goals outlined above yet has poor overall performance is relatively easy. However, such a machine is of questionable research value since its performance would be far lower than that of a comparable hardwired machine and any performance data or tradeoffs observed would be, in general, inapplicable to hardwired implementations of the same architecture.

3.1.2 Programmable Node Controller

Supporting multiple protocols is difficult with a fixed, hardwired node controller. At least some parts of the system must be flexible (that is, programmable), so that the same underlying hardware can be used to implement multiple protocols. A programmable protocol engine, which allows the protocol to be tailored to the researcher’s needs simply by changing the code the programmable protocol engine executes, provides this needed flexibility. The engine’s ability to support multiple protocols also allows support for both message passing and cache-coherent shared memory, again by writing the appropriate code for the protocol engine. As noted earlier, one goal of FLASH is to provide integrated support for these communication paradigms; the programmable engine allows this support to be provided without implementing separate, dedicated message passing and cache coherence hardware. Equally important is the ability to support a range of performance monitoring facilities; these can be provided by inserting the appropriate performance data collection code into the normal protocol code. This approach allows the performance data collection to be specialized to the protocol in use and permits the type and level of detail of this data to be changed.

Even ignoring the goal of supporting multiple protocols, the use of a programmable engine offers two other advantages: it makes the system easier to debug and it may reduce design time. Debugging is eased because protocol bugs can be fixed even after the machine has been built, and because software workarounds for hardware bugs are possible even at the lowest levels of system operation. Also, the same mechanisms used to collect performance data can be used to detect errors in system operation (“assert” statements in protocol code, for example) and to collect, filter, and analyze debugging information that later can be exported to the designers. Design time may be reduced because much of the complexity of implementing a protocol can be moved into software, thereby potentially reducing the overall amount of hardware to be designed, implemented, and tested, and reducing the risk that undetected protocol bugs will necessitate chip respins.

Finally, a programmable node controller offers advantages for improving the reliability of large-scale systems. Recovery from component failure is very difficult in a shared memory system because application and operating system state tends to be dispersed throughout the machine. Complex algorithms are needed to perform coordinated recovery among the operational processing nodes remaining after a failure. The availability of a programmable node controller allows some of this recovery code to be executed on the node controller itself, which is often a significant advantage since the memory system state and other very low-level information is directly accessible to the node controller. Further discussion of this topic is beyond the scope of this dissertation; Rosenblum [87] covers the subject in detail.

Because a programmable protocol engine can satisfy the FLASH design goals noted above while simultaneously potentially simplifying system implementation and improving reliability, the first FLASH design decision was to use such an engine as the core of the FLASH node controller.

3.1.3 Logic Overhead Reduction

A second goal of FLASH is to overcome, or at least reduce, the logic and memory overhead — estimated at 20% for DASH — associated with implementing cache coherence. In DASH this overhead had two main sources. First was the additional logic required to add support for distributed cache coherence to an existing bus-based multiprocessor system. Necessary changes in DASH included the addition of the directory controller and reply controller, which together implemented all protocol and directory processing, as well as relatively minor modifications to the existing CPU interface logic, memory controller, and bus arbitration logic to support operations, such as forced bus retries, required in a distributed multiprocessor but not in the original bus-based system.

FLASH reduces logic overhead in two ways. First, in DASH, the memory controller, system bus control and arbitration logic, and I/O controller were implemented using MSI technology on a collection of several printed circuit boards. By taking advantage of progress in integrated circuit technology and the capability and capacity of ASICs, FLASH implements all this functionality on a single chip, namely the MAGIC node controller. Although this reduction of several printed circuit boards to a single node controller chip has obvious cost, performance, reliability, and packaging density advantages, it does not, in itself, reduce the logic overhead of cache coherence, since any contemporary bus-based multiprocessor system, or even a uniprocessor system, would use a similar approach. However, FLASH goes one step further and integrates all protocol processing on the node controller as well.

This integration of the memory controller, I/O controller, bus controller, and other logic necessary for any bus-based multiprocessor or uniprocessor on the same chip as the additional protocol processing logic needed to add support for distributed cache coherence is the key to reducing the logic overhead. Whereas in DASH the addition of two extra printed circuit boards was needed to add distributed cache coherence support, FLASH has no visible additional hardware. Instead, the additional functionality is integrated into the same node controller chip that would be required even if the underlying system were just a bus-based multiprocessor or even a uniprocessor. Thus, FLASH reduces logic overhead in two ways: by taking advantage of integrated circuit technology to implement on a single node controller chip logic that required several printed circuit boards on DASH and by adding protocol processing functionality to this chip as well.

Even with the single-chip approach, the logic overhead of cache coherence in FLASH is not zero. Adding protocol processing functionality to the node controller complicates its design and adds to the overall size of the chip, increasing both the raw gate count and die area. The important distinction is that the overhead in FLASH is a marginal overhead, added to a component that is already needed in the system. As a result, although the initial node controller chip design effort is larger, the ultimate difference in chip cost is likely to be minimal and, in any event, will be less than the cost of adding additional, separate chips to the processing node. Further reducing overhead, the same protocol engine that was added to the node controller to implement distributed cache coherence also can be used to manage communication among the components within the local processing node. In this way, the addition of the protocol engine eliminates the need for some of the control and sequencing logic that would have been required in the bus-based or uniprocessor version of the node controller.

The combination of memory controller, I/O controller, and protocol engine also allows a very close coupling of the compute processor, local memory, local I/O subsystem, and interconnection network, thereby improving overall system performance in addition to minimizing logic overhead. Because of the functionality integrated on the node controller, and because the protocol engine is an addition to a chip that is needed anyway, quantitatively estimating the logic overhead of implementing cache coherence in FLASH is difficult. In contrast, DASH's directory controller and reply controller were physically separate printed circuit boards, and their gate count (or IC count) could be used directly to compute the logic overhead.

3.1.4 Memory Overhead Reduction

The second source of overhead in DASH was memory overhead resulting from the use of dedicated directory storage. DASH's bitvector directory structure requires a 20-bit directory entry for each 16-byte (128-bit) cache line in a node's memory, resulting in a directory memory overhead of (20 / 128), or approximately 15%. Additional memory overheads include the remote access cache (RAC), with a total size of 176KB, and the dedicated trace buffer used to store performance data, with a total size of 8MB. Unlike the directory memory, the size of the latter two structures is constant; it does not vary with the size of the node's local memory.

FLASH seeks to overcome these memory overheads by eliminating the distinction between protocol storage and memory used for the operating system and applications. Each FLASH node provides a single main memory that is used both to store directory information and other protocol state and code for use by the protocol engine, as well as to store the traditional operating system and application data for use by the compute processor. Of course the use of a single memory does not actually eliminate the directory memory overhead present in DASH. If FLASH were to use the same 20-bit directory entry format as DASH did, it would require an equivalent amount of directory storage space. The difference is that this space would be allocated from a resource already needed on the processing node; no additional, dedicated memory is required. In addition, the fraction of main memory allocated for protocol processor use can vary dynamically. This capability is crucial given FLASH's support for multiple protocols; without it, an amount of memory large enough to store the largest protocol data structure of any anticipated protocol would have to be provided by additional, dedicated memory or reserved permanently in the node's main memory.

Just as important, the ability to dynamically change the fraction of main memory reserved for protocol engine use allows performance data, as well as protocol state, to be stored in main memory, thereby overcoming the limitations of DASH's fixed-size trace buffer. Also, the protocol processor and compute processor each have access to both protocol storage as well as operating system and application data structures, facilitating the sharing of information and easing the interaction of the operating system and protocol engine. Overall, dynamically using a fraction of main memory for protocol storage and performance data avoids arbitrary limits on the size of these structures — all of main memory could be used if desired — and increases overall system flexibility.

In addition to eliminating dedicated protocol storage, a further means of reducing memory overhead is to use more scalable directory data structure formats for implementing cache coherence. Chapter 5 discusses this topic in detail, but the basic problem with the simple bitvector directory scheme used in DASH — essentially the same format illustrated in Figure 2.4 — is that the aggregate size of the directory grows in proportion to the square of the system memory size and processor count. For the 64-CPU maximum DASH processor count, the bitvector overhead is tolerable. FLASH aims to support far larger processor counts, however, and for larger systems the overhead of a bitvector directory format is prohibitive. As a result, a second aspect of reducing memory overhead is allowing a more scalable and memory-efficient directory format to be employed.

3.2 FLASH Node Architecture

With the goals outlined above in mind, the next step was to develop the FLASH processing node architecture. FLASH is a scalable system and its node architecture, depicted in Figure 3.1, is very similar to the generic distributed multiprocessor node shown in Figure 2.2. Each FLASH node contains a MIPS R10000 compute processor [113] with its second-level off-chip cache, a portion of the machine's memory, a connection to the local PCI-based I/O subsystem [81], a connection to the interconnection network, and the node controller, called MAGIC (Memory And General Interconnect Controller). The MAGIC node controller is responsible for handling the low-level requests generated by the local R10000 and local I/O subsystem, as well as the low-level requests that arrive from other processing nodes via the interconnection network. These low-level requests will be called *messages* in the remainder of this document. The interconnection network ties together a collection of FLASH nodes to form a complete FLASH system, as illustrated in the left-hand portion of the figure.

Although similar to the generic distributed multiprocessor node, the FLASH node differs in two ways, as described earlier. First, the MAGIC node controller uses a flexible, programmable engine rather than a set of hardwired state machines to implement the system communication paradigm. Second, the FLASH node contains no dedicated protocol storage, such as for a directory. Instead, all protocol state information is kept in a portion of the node's local memory.

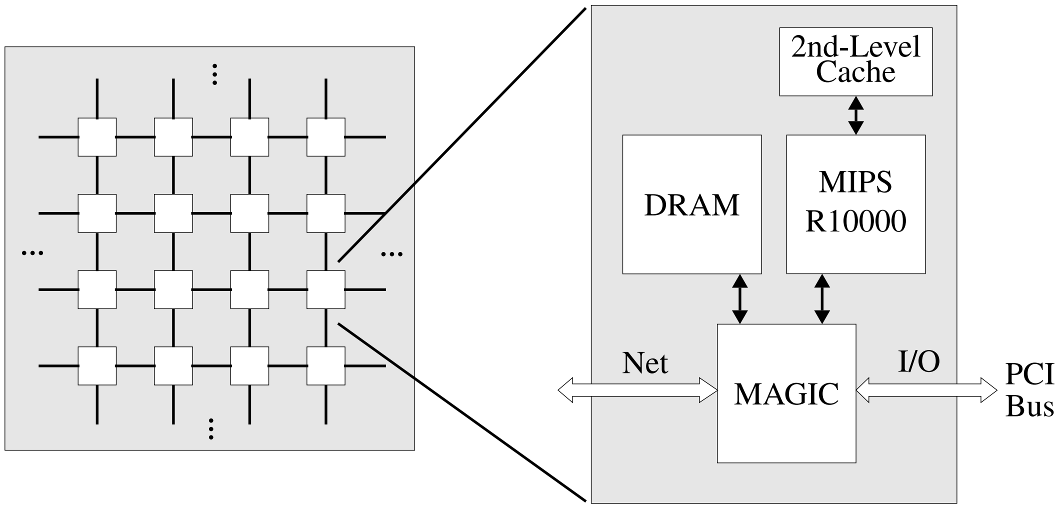


Figure 3.1 FLASH node architecture

3.3 Interconnection Network

To maximize system performance, the interconnection network must provide high bandwidth — comparable to the bandwidth available between the local memory and compute processor — and low inter-node message latency. At the same time, the network must be able to connect physically separate processing nodes that span both multiple printed circuit boards and multiple cabinets. Providing a high-bandwidth, low-latency link across physical distances of several meters is a significant logic and circuit design challenge in itself. While FLASH needs a reliable, scalable, high-performance interconnect, it is not the primary focus of the FLASH system design. For this reason, FLASH uses a commercial interconnection network, specifically the same CrayLink interconnection network used in the Silicon Graphics Origin 2000 system [64]. Use of a commercially-developed network provides tremendous leverage of industrial engineering expertise, eliminates the need to design and implement the network locally, and offers a reliable, tested means of connecting the processing nodes.

The CrayLink network is based on the Spider routing chip developed at Silicon Graphics [34]. Each Spider router implements a 6-port crossbar with internal buffering and flow control mechanisms. Figure 3.2 shows the external ports of a single Spider router. Each port on the router is bidirectional and provides 800 MB/s bandwidth in each direction,

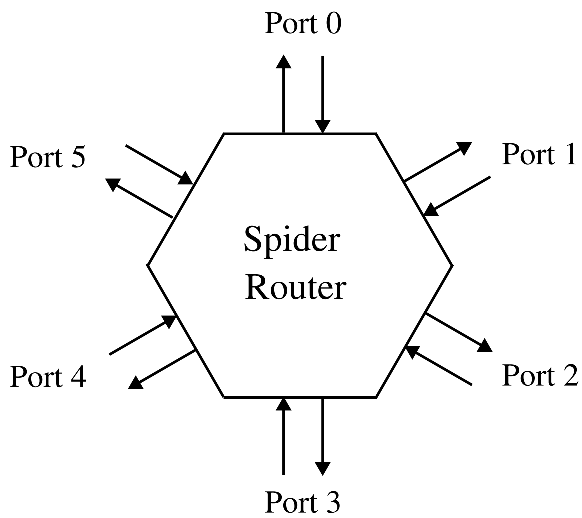


Figure 3.2 Spider router chip external interfaces

yielding an aggregate bandwidth of 9.6 GB/s through the router (six ports with two 800 MB/s directions per port). The internal crossbar within the router allows all six ports to be active simultaneously. Because network congestion can require a router to limit the amount of data it sends to a neighbor, the router provides internal buffering to handle the necessary flow control.

A complete interconnection network consists of a collection of routers connected to each other and to the MAGIC chips on the processing nodes. Figure 3.3 depicts a simple network used to connect eight processing nodes. The boxes labelled “R” in the figure represent individual routers; the boxes labelled “N” are FLASH processing nodes. Two processing nodes are connected to each router, and the routers themselves are then connected to provide the necessary connectivity among the nodes.

FLASH uses the same hypercube topology as the Origin 2000. Figure 3.3 shows how eight processors, and thus four routers, would be connected to form a two-dimensional hypercube. This processor count is so small that only four of the six ports on each router are needed. Larger systems maintain the hypercube topology, all the way to a 32-CPU system that uses all six ports on each of 16 routers to form a four-dimensional hypercube. Systems larger than 32 CPUs require more than six router ports and cannot be implemented directly. Instead, the network topology becomes a fat hypercube, as described in [34].

Messages are sent from a source node to a destination node, or more precisely from the MAGIC chip on the source node to the MAGIC chip on the destination node. A message

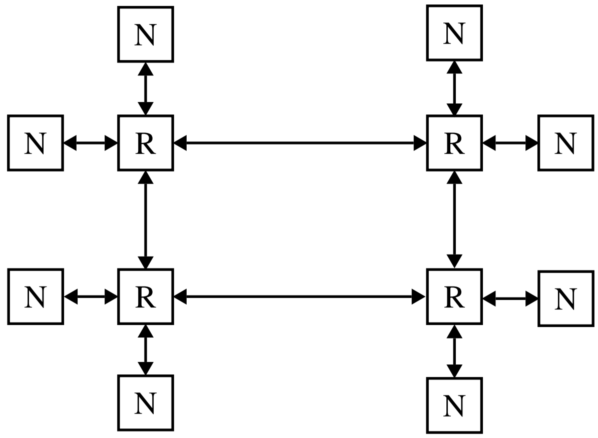


Figure 3.3 Eight-processor interconnection network topology

can be of arbitrary total length. However, for purposes of transferring the message through the interconnection network, the message is fragmented into 128-bit *micropackets* (also called flits) as shown in Figure 3.4.

All messages begin with a single header micropacket, which is followed by zero or more data micropackets. The header micropacket contains routing information specifying the path the message should take to get from source node to destination node as well as

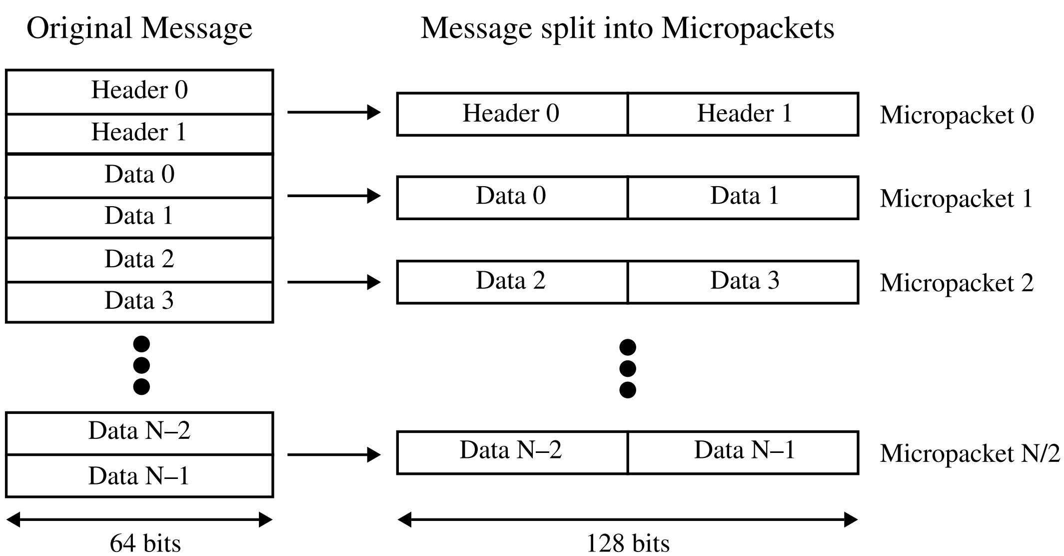


Figure 3.4 Message fragmentation into micropackets

additional fields for use in interpreting the message contents at the destination node's MAGIC chip. The data micropackets contain any data associated with the message. In low-congestion situations in which micropackets do not need to be buffered, a micropacket propagates through a single router in 50ns. Even with several routers between source and destination nodes, the overall transit time is low, an important factor in maintaining good system performance, particularly for cache-coherence protocols.

The router provides several features to increase network performance and reliability. Even though each router port is a single physical link, the router time-multiplexes four virtual channels [28] on the link, giving the appearance of four independent networks. At least two independent networks are necessary to provide deadlock-free routing for a typical cache-coherence protocol [66]; virtual channels provide these independent networks without incurring the hardware overhead of physically duplicating network links. To reduce transit time the router implements wormhole routing [29], meaning that micropackets are forwarded from one router to the next without waiting for the entire message to arrive. This technique effectively pipelines message transfer through the network and reduces end-to-end message delivery latency.

Link-level CRC codes aid reliable message delivery. These codes are computed as a micropacket exits a router (or MAGIC chip) and checked as the micropacket is received at the opposite end of the link. Micropackets corrupted while being transmitted across a link are detected by an incorrect CRC at the receiver, which requests that the micropacket be retransmitted until it arrives intact. This link-level error checking and retransmission mechanism uses a go-back-N sliding window protocol to maximize bandwidth while still providing error-free micropacket delivery at the link level.

3.4 System Packaging

Like the interconnection network, the system packaging — the cabinets, node board, and other physical enclosures — is an important component of the system implementation but was not a focus of the FLASH design effort. Designing a compact cabinet for the node board while providing sufficient cooling, power, and space for disk drives and other I/O devices is a significant mechanical and electrical engineering task. To minimize the amount of work the FLASH team needed to do in this area, the FLASH system packaging leverages heavily from existing Silicon Graphics products.

The system enclosure, containing the node board, power supply, fans, and space for PCI devices such as disk drives and graphics cards, is identical to the one designed for the SGI Origin 200 deskside system. The node board fits into the center of the cabinet and contains all components shown in Figure 3.1 with the exception of the R10000 and its second-level cache; these are provided on a separate daughtercard containing the R10000 processor, 1 MB of second-level cache, and a heatsink. As depicted in Figure 3.1, the MAGIC chip implements a direct connection to a PCI bus and can support up to four PCI devices on this bus. The node board itself contains an SGI-developed I/O chip (IOC3) that implements 100 Mbps Ethernet, fast SCSI, a serial port, and connections for a keyboard and other slow I/O devices. This chip consumes two PCI IDs. The Origin 200 cabinet provides an additional two full-length PCI slots, allowing standard PCI cards to be added to the system. These cards could include graphics controllers, additional SCSI controllers, or high-end networking cards, such as an FDDI card. Also in the cabinet is space for eight 3.5-inch disk drives, all connected to the IOC3's SCSI bus.

The node board design also is based on the one used for the Origin 200 motherboard. However, the FLASH version of the node board incorporates several changes — most notably the replacement of the Origin 200 node controller chip with the MAGIC chip — so a new node board layout was required. The FLASH node design includes a local memory system implemented using synchronous DRAM (SDRAM) DIMMs. Up to six DIMMs can be accommodated on a node board, allowing a total memory size of 192 MB with 16-Mbit SDRAM parts and 768 MB using 64-Mbit parts. The SDRAMs connect to a 128-bit bus operating at 50 MHz. This bus is then multiplexed on the board into a 64-bit 100 MHz bus that connects directly to MAGIC. Other major components include the MAGIC chip itself, the off-chip MAGIC data cache, and the "XC" chip, which converts the single-ended network signals from MAGIC into differential signals for transmission off the node board.

The node board does not contain a Spider router chip. Instead, dedicated routing cabinets provide the interconnection network. These so-called metarouters, developed by SGI, each contain eight Spider router chips connected in a three-dimensional hypercube. Node boards and metarouters are connected to form the complete interconnection network. Systems with 16 or fewer processors require just one metarouter; as the system size grows, additional metarouters are required.

3.5 Node Controller Implementation Options

The previous sections have presented the FLASH node in its final form, including the single-chip MAGIC node controller. A large focus of the FLASH system design effort, however, was the development of the node controller architecture. This section traces the evolution of the MAGIC architecture, from the initial architectural and performance constraints to the rationale behind the decision to implement a single-chip, integrated node controller.

For the reasons outlined in Section 3.1, the FLASH system uses a programmable node controller. The challenge was to insure that the use of a flexible, programmable node controller did not unacceptably reduce overall FLASH system performance. In particular, as described in Section 3.1, the goal was to achieve overall system performance competitive with that of a fully-hardwired machine having the same underlying system architecture. Evaluating this goal involved determining whether a programmable protocol engine could achieve competitive performance at all. If so, what features must this programmable engine include? And, aside from the programmable engine itself, what additional hardware support was needed? To determine if competitive performance was achievable, a study was undertaken that compared the performance of several programmable engine-based design alternatives to the estimated performance of a fully-hardwired design. The fully-hardwired design was assumed to be a system that used the FLASH system architecture but implemented the DASH protocol completely in hardware using the same technology constraints as FLASH.

Although Figure 3.1 depicts the node controller as a single entity, this view is an architectural perspective rather than an implementation necessity. In DASH, the node controller was implemented with many MSI and LSI parts on two large printed circuit boards. FLASH seeks to greatly reduce the raw node controller parts count, and the MAGIC node controller ultimately was implemented as a single chip. For the initial implementation studies, however, the node controller was not constrained to be a single chip and several implementation options were considered.

3.5.1 Trial 1: Off-the-Shelf Processor

The most straightforward approach to implementing a programmable node controller is to use an existing off-the-shelf processor. The primary advantage of this approach is its leverage of commercial design, engineering, and implementation expertise. A commercial CPU would undoubtedly run faster than any processor developed specifically for FLASH, while also costing less and helping to minimize the FLASH system design time. A commercial processor also implements a well-understood programming environment supported by a rich set of code development tools. In fact, one option is to use the same processor as the FLASH node's compute processor, an approach similar to the one adopted for the Intel Paragon system.

Despite these advantages, using an off-the-shelf processor presents several problems. First, such a processor would not interface efficiently to the other components of the processing node illustrated in Figure 2.2. The programmable engine, whether implemented as an off-the-shelf processor or not, provides only one part of the node controller functionality. Also needed is a memory controller, an I/O controller, a means to interface to the node's compute processor, and an interface to the interconnection network. Typical off-the-shelf processors provide none of these capabilities and, even if they did, a truly "glueless" interface would be unlikely.

The second problem is that an off-the-shelf processor is not adept at executing protocol code. Off-the-shelf processors, particularly ones of the same generation as the node's R10000 compute processor, are designed to execute application programs. But the characteristics of protocol code are very different from those of typical application code. Protocol code sequences tend to be very short, often only a few or few tens of instructions. Worse, these sequences always begin with a dispatch operation — the incoming request must be inspected, the address of the corresponding code sequence computed, and control transferred to this address. In addition, protocol state data structures typically consist of packed arrays of bitfields. The 20-bit DASH directory entry, for example, consists of one state bit, 16 presence bits, one zero vector bit, and two parity bits. In contrast, typical application code tends to consist of long, loop- and load/store-oriented code sequences that operate on individual data items that are at least byte-sized and byte-aligned. Hence, protocol code is more similar to the lowest-level interrupt vectoring and processing code in the operating system than to typical application code, and the protocol engine must be adept at handling interrupt-style code if it is to achieve good protocol processing performance.

Unfortunately, modern processors fare poorly at handling short, interrupt-style code sequences. These processors employ dynamic scheduling, out-of-order execution, deep pipelines, extensive branch prediction with large penalties for mispredictions, and other techniques aimed at improving the throughput of typical application code. While these techniques result in significant performance improvements on application code, they also render the design unsuitable for use as a protocol engine. On the R10000, for example, an uncached load — as would be used to retrieve data from the interconnection network or poll the status of an I/O device — requires over 15 cycles to complete. Given that the basic nature of protocol code is to interact with the interconnection network and the other components on the node board, such long-latency communication primitives are unacceptable. Even a more interrupt-driven interface would not help: an R10000 can require as many as 20 cycles to recognize an interrupt and begin executing the interrupt vectoring code.

Somewhat ironically, then, the best protocol processor is likely to be a previous-generation processor, such as the MIPS R4000 [57] embedded processor or R3000 [57]. The attractiveness of a previous-generation design lies not in its raw processing power — which is inferior to contemporary offerings — but in its efficient, direct interfaces to external components and its simpler internal pipeline structure that leads to faster interrupt response and lower uncached operation latencies and branch misprediction penalties. In addition, previous-generation processors are considerably cheaper than the state-of-the-art CPUs. A MIPS R10000, for instance, costs approximately $2000 as of this writing, whereas an R3000 costs just $40.[1] Therefore, the best candidate for an off-the-shelf protocol processor is a previous-generation processor and, as a result, the R3000 was selected for evaluation. However, because the FLASH system uses 64-bit addresses, the R3000 is assumed to be an updated version of the original architecture that retains the same pipeline, instruction set, and other architectural features but widens the internal registers, datapaths, and external system bus to 64 bits from the 32-bit size of these elements in the conventional R3000 implementation.

3.5.1.1 Node Controller Organization

To determine whether this off-the-shelf R3000 processor could yield good overall protocol processing performance despite the problems outlined above, key parts of the DASH protocol were coded in MIPS R3000 assembly language and the resulting code sequences examined. Generating the proper code sequences requires some assumptions about the underlying node controller hardware and its connections to the rest of the processing node; Figure 3.5 illustrates the basic node controller architecture.

As the figure shows, the node controller consists of an R3000 processor and its system interface to external instruction and data caches as well as to the additional processing node components. Surrounding the R3000 are additional interface logic elements to provide the interfaces to the local compute processor, network, memory system, and I/O subsystem. The interface logic elements are drawn as ovals in the figure to emphasize that their detailed implementation is unspecified. Rather, the initial assumption is that the interface logic elements are just simple registers and state machines that provide a reasonable logical interface to the node components without providing any significant processing capability of their own. The memory controller interface element, for example, might provide a register to specify an address to read or write and an additional set of registers to hold the data to be read or written, along with the necessary state machine to perform the DRAM control sequencing. Updating and monitoring the contents of these registers is the

1. The R10000 price refers to the NEC µPD30700RS part. The R3000 price refers to the NEC µPD3010GC, which is a 64-bit processor with an R3000-style five-stage pipeline and a 64-bit external system bus. Both prices were quoted by Marshall Electronics for quantity 100 purchases.

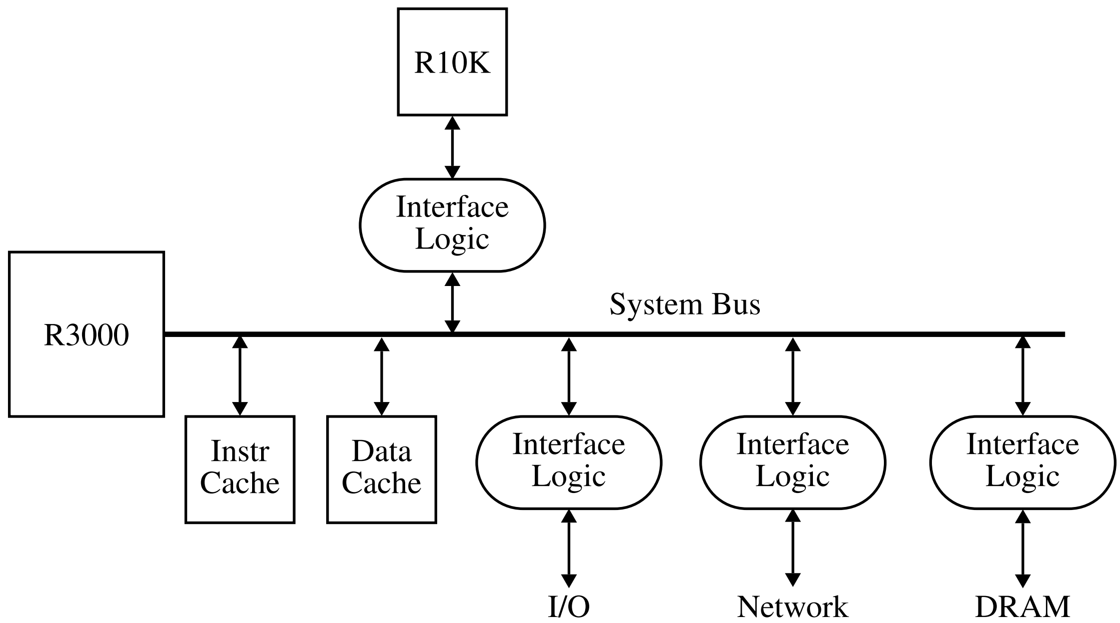


Figure 3.5 Processing node with off-the-shelf R3000 protocol engine

responsibility of the R3000 node controller, which uses normal load and store instructions to read and write the memory interface registers. That is, the memory interface registers are memory-mapped into the R3000's address space, as are the other interface logic elements.

3.5.1.2 Message Dispatch Loop

Internal operation of the R3000 protocol engine consists, at least conceptually, of a single `while` loop whose structure is given in Figure 3.6. Processing of an incoming mes-

```
while (TRUE) {            /* Loop forever */
   switch (msg.type) {   /* Dispatch on incoming message type */
      case READ:
         /* Code to process READ operation */
      case WRITE:
         /* Code to process WRITE operation */
      case INVALIDATE_REQUEST:
         /* Code to process INVALIDATE_REQUEST operation */
      case ...
   }
}
```

Figure 3.6 Basic incoming message dispatch and processing loop sage begins with the `switch` statement, which performs a dispatch based on the incoming message type. Once the dispatch completes, control transfers to the code sequence appropriate to the message type. This code sequence, or "handler," is typically very short, often only ten or a few tens of instructions. The handler is responsible for updating any necessary protocol state (for example, the directory), for initiating any necessary memory operations, for transferring data, and for composing and sending any outgoing messages generated in response to the incoming message. Interacting with the memory and sending outgoing messages requires load and store operations to the proper logic interface elements, which are memory-mapped into the R3000 address space as described above.

Evaluation of the DASH protocol code sequences in this environment revealed three performance bottlenecks: dispatch overhead, data transfer, and bitfield operations.

3.5.1.3 Dispatch Overhead

Although transferring the message on-chip and storing the message header in a processor register may add secondary overheads, the primary dispatch performance bottleneck occurs in processing the switch statement at the start of the message dispatch loop. To conserve storage space and network bandwidth, the message header, containing information such as the message type, source processor, address, and other control information, will consist of several bitfields packed into a single register. One of these bitfields is the message type. Hence the first step in the implementation of the `switch` statement is to extract the message type from the other fields of the message header. Next, the message type must be used to access a software jump table, whose base address is assumed to be stored in one of the R3000's registers and whose contents are a contiguous array of 64-bit program counter values indexed by incoming message type. At the assembly level, then, the `switch` statement is implemented as the code fragment of Figure 3.7.

```
switch:
   andi  rX,rH,0xff     /* Mask off message type from header in rH */
   sll   rX,rX,3        /* Scale by size of jump table entry (64 bits) */
   addu  rX,rX,rB       /* Add to base address of jump table in rB */
   ld    rX,0(rX)       /* Load handler's starting PC value */
   nop                  /* Load delay slot */
   jr    rX             /* Jump to start of handler */
   nop                  /* Jump delay slot */
handler:
   ...                  /* First instruction of handler */
```

Figure 3.7 Dispatch assembly code

This code assumes the message header is loaded automatically into register `rH` and that the message type occupies the lower eight bits of the header, so as to facilitate extracting the message type using a single `andi` (logical and) instruction.

Although execution of the dispatch code requires just seven cycles, the code represents a significant processing overhead when measured against the 10 or 20 instructions in a typical handler. Also, the most frequently executed handlers, such a cache miss to a local address, tend to be the simplest and thus shortest, exacerbating the effect of the seven-cycle dispatch. Furthermore, the five actual instructions in the dispatch fragment are serially dependent, meaning that changing to a more aggressive, multiple-issue protocol processor architecture would not reduce the dispatch overhead; in fact, the relative overhead of dispatch increases in this case since the dispatch code requires the same number of cycles yet the handler code probably requires fewer. Some optimizations can reduce the length of the dispatch. A register-plus-register addressing mode for the `ld` instruction allows the `addu` instruction to be eliminated, for instance, though the gain would be minor. Filling the load and jump delay slots also would help, and these slots are best filled with the first two instructions from the most common handler. Of course, at this point in the code the eventual handler has not even been identified yet, so choosing the proper two instructions would be difficult.

3.5.1.4 Data Movement Overhead

The second overhead of using the off-the-shelf R3000 is data movement. Most incoming messages require some kind of data movement to occur. The most common case is a cache miss received from the local compute processor, which requires that a cache line of data be read from the node's local memory and transferred to the compute processor. Because of the assumed simplicity of the logic interface elements, all data transfer must be provided explicitly by the handler code. A memory read, for example, begins with the handler storing the address into the proper register in the memory interface element and initiating the memory read. Once the memory read completes, the handler must read each data word from a register in the memory interface element and then store the data word to a register in the compute processor interface element. The FLASH system, as noted in Section 3.2, uses 128-byte cache lines. As a result, transfer of a cache line of data requires 16 64-bit loads from the memory interface element and 16 corresponding stores to the compute processor interface element, for a total of 32 handler instructions devoted just to data movement. Clearly this overhead is unacceptable, and can be reduced significantly with a modest amount of DMA-like support logic. However, the goal of the off-the-shelf protocol processor was to keep the external logic as simple as possible; the resulting need to implement all data transfer in handler code identifies a serious performance bottleneck inherent in this approach.

3.5.1.5 Bitfield Manipulation Overhead

Bitfield manipulation is the final overhead of using the R3000. The MIPS instruction set does not provide any native support for bitfield manipulations.[2] Instead, these operations must be synthesized from shift and logical instructions. Typical application data structures contain few bitfields, so the lack of instruction set support has only minimal impact on the performance of application code. Protocol state data structures, however, almost always consist of several bitfields packed into a single 64-bit word. The format of FLASH message header, to be presented in Section 4.3, for example, is shown in Figure 3.8.

63 60	59 56	55 55	54 52	51 40	39 28	27 16	15 14	13 12	11 0
PBUF	WBUF	DBL	ENB	MISC	DEST	SRC	Zero	LEN	TYPE
4	4	1	3	12	12	12	2	2	12

Figure 3.8 Bitfields in FLASH message header

Putting aside the meaning of the fields in the header, the key point is that the header is a collection of 10 individual bitfields, all packed into a single 64-bit word. Typical directory entries have a similar format. The code to perform a register-to-register bitfield extract operation consists of two instructions; the code to perform a register-to-register bitfield insert consists of five instructions. Because bitfield manipulation is such a common operation, the multicycle bitfield operations are a significant overhead.

3.5.2 Trial 2: Off-the-Shelf Processor with External Hardware Support

The evaluation presented in the previous section revealed three overheads in the use of an off-the-shelf protocol processor: dispatch, data transfer, and bitfield manipulation. The

2. Other instruction sets, most notably the PowerPC and Motorola 680x0, do provide bitfield instructions. The 68K-based products tend to be too slow overall to compete with the R3000, even given the R3000's lack of bitfield support. A PowerPC processor is a reasonable alternative to the R3000, but its bitfield instructions are still insufficient for FLASH needs — it has no branch-on-bit instructions, for example — and, in any event, a more cohesive design results if the node controller uses the same instruction set as the MIPS R10000 compute processor.

result was a system that had very poor protocol processing performance even though it effectively leveraged commercial CPU implementation expertise to reduce FLASH design time. As Section 3.1 notes, the goal of FLASH is to achieve overall system performance competitive with a hardwired machine; the use of an off-the-shelf R3000 as the protocol processor, at least with the simple external logic elements outlined above, would not have achieved this goal. Yet of the three overheads, only bitfield manipulation is inherent to the R3000 instruction set; both dispatch and data transfer are amenable to external hardware acceleration. For this reason, the next design evaluation focused on the same R3000 protocol processor as used in Trial 1, but with more sophisticated external interface logic. In particular, an external dispatch table was added, as was external support for data transfer.

The dispatch logic was a RAM-based table with some support logic. Incoming messages flowed through the dispatch logic, which extracted the message type and used it to index into the RAM, yielding the handler's starting program counter value. The program counter was stored in an external register, allowing the R3000 to read from this register and jump to the resulting address, thereby reducing the dispatch fragment to a single load and jump, each with one delay slot, resulting in a four-cycle dispatch rather than seven, almost a 50% reduction. A further reduction could be achieved by placing the load from the program counter register at the end of every handler, allowing the load delay slot to be filled with the final handler instruction and reducing the effective dispatch overhead from four cycles to three.

The data transfer facility was provided in the form of a simple DMA engine, again implemented in logic external to the R3000, that could transfer data among the local compute processor, memory, network, and I/O subsystem. The handler code was responsible for configuring and initiating the data transfer — specifying the source interface and destination interface, for instance — but once underway, the data transfer logic managed the actual movement of data words.

Overall, this approach is similar to the one proposed in Trial 1 (Section 3.5.1) in that it combines an off-the-shelf protocol processor with external logic. In contrast to Trial 1, however, the external logic is now much more capable and more specifically targeted at alleviating the dispatch and data transfer overheads identified in Trial 1. Unfortunately, although this approach certainly has better performance, it still suffers from the lack of bitfield support and, perhaps more important, from the need to implement external hardware. The need for external hardware is troublesome in that it is fairly complicated yet needs to run quickly. Even implementing the simple external logic elements proposed in Trial 1, much less the significantly more complex external logic proposed for Trial 2, would require a non-trivial amount of hardware. Not only would this hardware have added to the design time and effort that the use of an off-the-shelf processor was meant to reduce, but it also would have increased the logic overhead of implementing the system's communication paradigm, contrary to the specific FLASH goal of minimizing this overhead.

Perhaps some of this external logic could be moved on-chip by using a commercial R3000-based microcontroller that implemented, for instance, a memory controller and DMA engine on-chip. However, no commercial microcontroller provides the exact interfaces needed for use in the FLASH node and so some external logic still would be required. At a more architectural level, external logic has the additional drawback of decreasing the coupling between the node controller and the other node components. In fact, the additional overhead of communicating with the node board components via the external interface elements was not even considered in the analysis above, yet close coupling between the node controller and node board components is critical to minimizing overall protocol processing overhead, regardless of the node controller choice.

3.5.3 Trial 3: Custom Processor Core with Integrated Hardware Support

Section 3.5.1 and Section 3.5.2 demonstrated that neither the off-the-shelf R3000 with simple external interface elements nor the same R3000 with more complex external interface logic yield acceptable protocol performance. Both approaches require external hardware and neither addresses the need for close coupling of the node controller and other node board components. In addition, the lack of bitfield support in the R3000 is ill-suited for manipulating protocol state data structures. The key to addressing these performance bottlenecks is to eliminate all external logic by providing the required functionality on-chip and to tailor the protocol processor's instruction set to the characteristics of protocol code. Such an integrated approach retains the essential flexibility of a programmable protocol engine yet also achieves the close coupling to the other node board components and the efficient protocol code sequences necessary to achieve competitive overall performance.

Specifically, the integrated node controller must incorporate several key architectural features. First was the need to separate control processing (that is, protocol processing) from data transfer and to provide dedicated logic for each task. This separation offloads the data transfer burden from the handler code and, just as important, allows data transfer and protocol processing to occur concurrently. Second was the need for a hardware dispatch mechanism similar to the one described in Section 3.5.2. Third, the protocol processor's instruction set, while based on the R3000 instruction set, must be extended to match the characteristics of protocol code. In addition to bitfield instructions, the protocol processor also needed instructions to allow efficient communication with the other functional units within the integrated node controller. This close coupling of the protocol processor to the internal node controller functional units must be extended off-chip as well by implementing direct, glueless interfaces to the node board components so that no external interface logic is required. Finally, though not discussed above, a second-order protocol processing overhead resulted from the need to perform operations in handler code that are inherently inefficient to perform in software. Examples include bit swizzling, data format conversions (such as the swapping of the source and destination node bitfields in the message header of Figure 3.8), range checks, flow control, etc. To reduce this overhead, additional hardware must be provided to perform these functions, again with the goal of offloading the protocol processor.

Equally important as specifying the necessary features was identifying those features present in an off-the-shelf processor that are unneeded for protocol processing. Elimination of these features simplifies the design of the protocol engine and reduces die area. For example, the compute processor issues only physical addresses on its system bus and all code executed on the protocol processor is privileged, so the node controller has no need for any hardware-based address translation or protection facilities nor any need to support virtual memory. The on-chip interface logic can be closely coupled to the protocol processor, and the protocol processor code can be designed to execute in a non-blocking fashion, thereby eliminating the need for general-purpose interrupt and exception support. Because protocol processor code is so stylized, static scheduling works well and eliminates the need for any dynamic instruction issue or execution support. In addition, floating-point operations are essentially absent in protocol code, so only integer execution units are required. Floating-point data types are useful in certain situations — for example, for a floating-point fetch&op primitive [36] — but cache coherence and message passing protocols do not use floating point, making software emulation of floating point the appropriate choice for the rare cases in which it is used.

A final challenge in the design of the integrated node controller was balancing the functionality and performance advantages of dedicated hardware versus the desire to support multiple protocols and maintain overall design flexibility. Dedicated hardware support, such as the dispatch mechanism, definitely is required to overcome fundamental performance bottlenecks in the use of a programmable protocol engine. But at the same time this hardware must not impact flexibility; that is, any additional hardware support must improve protocol processing performance in general, and not just on a particular set of protocols. The additional hardware also must not compromise the node controller's support for protocols different from the ones that motivated the addition of the hardware in the first place.

Dispatch, data transfer, and bitfield overheads are common to almost every protocol, so the corresponding support hardware added to the node controller satisfies the above criteria. However, the dispatch logic in particular must be implemented carefully if it is to be useful for a variety of protocols rather than for just a few. For example, while it is reasonable (and essentially necessary from a hardware design perspective) to fix the position and size of the message type field within the FLASH message header, it is not reasonable to fix how the bits of the message type field will be used. The dispatch logic implementation might be simplified, for instance, if the message type encodings for the most common cache coherence messages were predefined. This approach is unacceptable, however, because it is protocol-specific, in this case to typical cache coherence protocols. Message-passing and synchronization protocols do not use the same message types as cache coherence protocols and the predefined message types are likely to be nonsensical when one of these protocols is in use. As a result, the dispatch logic could not be used efficiently and may even degrade performance for non-cache coherence protocols. Careful design of the dispatch logic, as well as the other dedicated hardware support logic, is required to insure that it improves performance yet is not protocol-specific and maintains the overall node controller flexibility.

A detailed design was performed of an integrated node controller with the internal architecture outlined above. The major components of this design and their connectivity are illustrated in Figure 3.9.

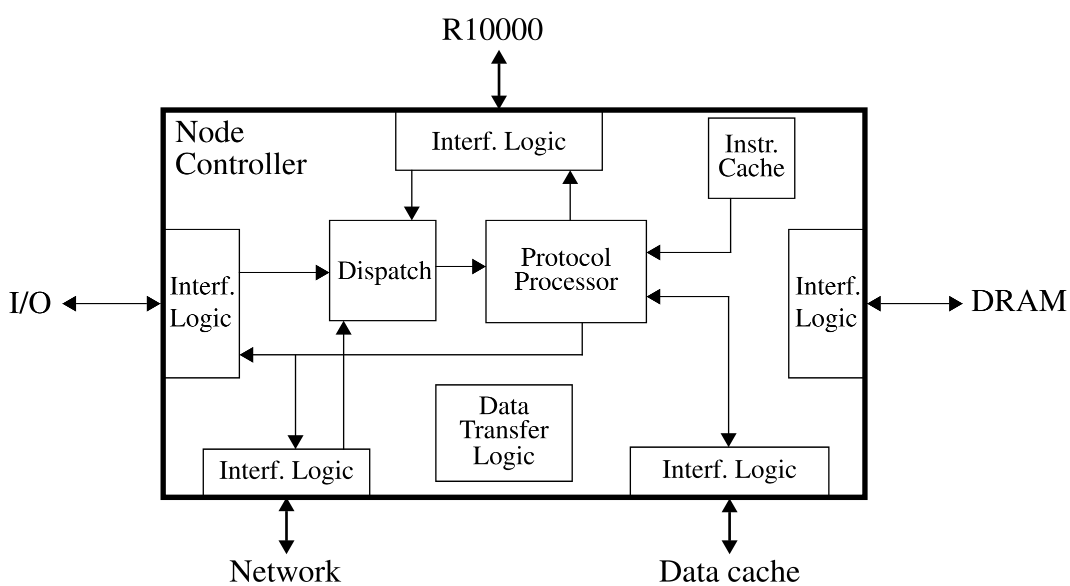


Figure 3.9 Integrated node controller architecture

The basic structure of this design evolved into the actual MAGIC architecture, to be presented in Chapter 4. A detailed performance study using FlashLite, the FLASH system-level simulator, compared the performance of the actual FLASH and MAGIC architecture to an "ideal machine" [44] that retained the same FLASH and MAGIC architecture but replaced the programmable protocol processor with a hypothetical version that processed all messages in zero time. The results of this study confirmed that the FLASH system performance was competitive with the ideal machine's over a range of scientific applications. In particular, for several applications drawn from the SPLASH-2 application suite [110], FLASH performance was within 12% of the ideal machine and even closer to the performance of a realistic hardwired implementation, an encouraging result.

3.6 Summary

The FLASH design addresses three goals: overcoming some of the limitations of the DASH design, reducing the hardware overhead associated with cache coherence, and providing a high-performance platform for exploring the design of large-scale distributed multiprocessors. The key component in achieving these goals is the use of a flexible, programmable node controller, the MAGIC chip. Most multiprocessor systems provide only a single communication paradigm and implement support for it completely in hardware in the node controller. MAGIC's programmable protocol engine allows it to support multiple communication paradigms, multiple protocols, and a variety of performance monitoring facilities. In addition, all protocol state, protocol code, and performance data is stored in a portion of the node's main memory rather than in additional, dedicated memory.

FLASH retains the basic characteristics and processing node structure of a distributed multiprocessor. Neither the interconnection network nor the system packaging was the focus of the FLASH design effort, however, and both were leveraged from commercial products developed at Silicon Graphics. The interconnection network, called CrayLink, is the same as used in the SGI Origin 2000. Its Spider router chips provide 800 MB/s bandwidth into and out of each FLASH node and connect all nodes using a hypercube topology. Low latency through each router reduces the end-to-end message delivery delay, and link-level error detection increases network reliability. The system packaging, including the enclosures, power supplies, cooling, and basic node board design, is the same as is used in the SGI Origin 200 workstation.

Flexible protocol processing with no fixed limits on the size of protocol and performance data structures are crucial elements in overcoming the limitations of DASH's hardwired cache coherence implementation and, more generally, in supporting distributed multiprocessor research. So, too, is the implementation of all protocol processing functionality on the same node controller chip as would be needed in a bus-based multiprocessor or uniprocessor. Integration of the protocol processor on the same chip as the memory controller and I/O controller minimizes the logic overhead of adding protocol processing support to the system and amortizes its cost.

Despite the advantages of a programmable protocol engine, FLASH's performance must be comparable to a machine with a similar architecture but using a full-hardwired node controller. Without competitive performance, research results gathered on FLASH lose their applicability to the design of similar, though hardwired, machines.

Achieving competitive system performance requires excellent protocol processing performance. An off-the-shelf protocol processor offers numerous advantages, mainly in its leverage of commercial processor design expertise and cost structure, but suffers from several overheads that greatly reduce its effectiveness as a protocol processor. The most significant of these overheads are dispatch, data transfer, and bitfield operations. Adding additional external hardware to alleviate some of these overheads can improve protocol processing performance, but not to a sufficient degree. External hardware also poses design challenges of its own, and leads to a lack of close coupling between the protocol processor and the other node components.

The only node controller design that can achieve the requisite performance level is a single chip that integrates not only the protocol processor itself but all support logic as well. Such an integrated solution allows the protocol processor and surrounding support logic to be optimized to the characteristics of protocol code and permits a very close coupling among all processing node components. The architecture of the MAGIC chip, presented in the next chapter, reflects this goal of an integrated node controller.

Chapter 4

MAGIC Architecture and Implementation

Chapter 3 described the evolution of the MAGIC design from the initial investigations of using an off-the-shelf processor to the eventual decision to design an integrated node controller to maximize protocol processing performance and minimize logic overhead. Whereas Section 3.5.3 presented the internal architecture of the integrated node controller in general terms, this chapter presents the architecture of the MAGIC chip in detail, starting with a high-level overview of the key architectural features and followed by a discussion of each of the major functional units. The goal is not to document the chip exhaustively; the MAGIC Specification [63] serves this purpose. Rather, the goal is to explain in some depth how the major components on the chip operate, how they cooperate to perform efficient protocol processing, and how the actual MAGIC design realizes the architectural goals outlined in Chapter 3.

4.1 MAGIC Architecture Overview[1]

The MAGIC architecture is designed to offer both flexibility and high performance. As described earlier, MAGIC includes a programmable protocol processor for flexibility. MAGIC's central location within the node ensures that it sees all processor, network, and I/O transactions, allowing it to control all node resources and support a variety of protocols. Also as noted, to avoid limiting the node design to any specific protocol and to accommodate protocols with varying memory requirements, the FLASH node contains no dedicated protocol storage; instead, both the protocol code and protocol data reside in a reserved portion of the node's main memory. However, to provide high-speed access to frequently-used protocol code and data, MAGIC uses an on-chip instruction and off-chip data cache.

FLASH nodes communicate by sending intra- and inter-node commands, referred to as *messages*. To implement a protocol on FLASH, one must define what kinds of messages will be exchanged (the *message types*), and write the corresponding code sequences for the protocol processor (the *handlers*). Each handler performs the necessary actions based on the protocol state and the information in the message it receives. Handler actions

1. The overview of MAGIC appeared previously in [62].

include updating protocol state, communicating with the local compute processor, and communicating with other nodes via the network.

Multiple protocols can be integrated efficiently in FLASH by ensuring that messages in different protocols are assigned different message types. The handlers for the various protocols then can be dispatched as efficiently as if only a single protocol were resident on the machine. Moreover, although the handlers are dynamically interleaved, each handler invocation runs without interruption on MAGIC's embedded processor, easing the concurrent sharing of state and other critical resources. MAGIC also provides protocol-independent deadlock avoidance support, allowing multiple protocols to coexist without deadlocking the machine or having other negative interactions.

Fundamentally, protocol handlers must perform two tasks: data movement and state manipulation. The MAGIC architecture exploits the relative independence of these tasks by separating control and data processing. Hardwired data movement logic achieves low latency and high bandwidth by supporting highly-pipelined data transfers without extra copying within the chip. The protocol processor manages state manipulation, employing a hardware dispatch table to help service requests quickly and a coarse-level pipeline to reduce protocol processor occupancy. This separation and specialization of data transfer and control logic ensures that MAGIC does not become a latency or bandwidth bottleneck.

As messages enter the MAGIC chip they are split into message headers and message data. Message headers flow through the *control macropipeline* while message data flows through the *data transfer logic*, depicted in Figure 4.1. Data and control information are recombined as outgoing message headers merge with the associated outgoing message data to form complete outgoing messages.

4.1.1 Data Transfer Logic

Both message passing and cache coherence protocols require data connections among the network, local memory, and local processor. Because the structure of these connections is protocol-independent, the data transfer logic can be implemented completely in hardware without causing a loss of overall protocol processing flexibility. The hardwired implementation minimizes data access latency, maximizes data transfer bandwidth, and frees the protocol processor from having to perform data transfers itself.

Figure 4.2 shows the data transfer logic in detail. When messages arrive from the network, processor, or I/O subsystem, the network interface (NI), processor interface (PI), or I/O interface (IO) splits the message into message header and message data, as noted above. If the message contains data, the data is copied into a *data buffer*, a temporary stor-

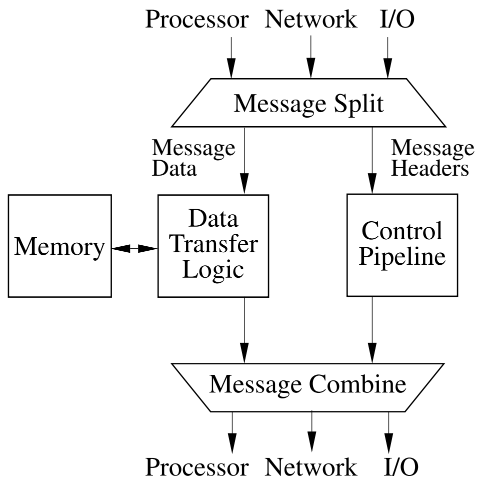


Figure 4.1 Message flow in MAGIC

age element contained on the MAGIC chip that is used to stage data as it is forwarded from source to destination. Sixteen data buffers are provided on-chip, each large enough to store one cache line.

Staging data through data buffers allows the data transfer logic to achieve low latency and high bandwidth through data pipelining and elimination of multiple data copies. Data pipelining is achieved by tagging each data buffer word with a valid bit. The functional unit reading data from the data buffer monitors the valid bits to pipeline the data from source to destination — the destination does not have to wait for the entire buffer to be written before starting to read data from the buffer. Multiple data copies are eliminated by

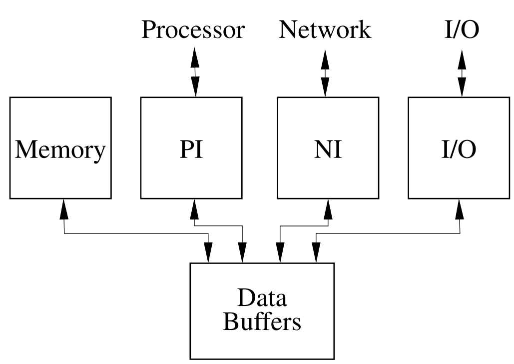


Figure 4.2 Data transfer logic

placing data into a data buffer as it is received from the network, processor, or I/O subsystem, and keeping the data in the same buffer until it is delivered to its destination. The control macropipeline rarely manipulates data directly; instead, it uses the number of the data buffer associated with a message to instruct the data transfer logic to deliver the data to the proper destination.

4.1.2 Control Macropipeline

The control macropipeline must satisfy two potentially conflicting goals: it must provide flexible support for a variety of protocols, yet it must process protocol operations quickly enough to ensure that control processing time does not dominate the data transfer time. A programmable controller provides the flexibility. Additional hardware support ensures that the controller can process protocol operations efficiently. First, a hardware-based message dispatch mechanism eliminates the need for the controller to perform message dispatch in software. Second, this dispatch mechanism allows a speculative memory operation to be initiated even before the controller begins processing the message, thereby reducing the data access time. Third, in addition to standard RISC instructions, the controller's instruction set includes bitfield manipulation and other special instructions to provide efficient support for common protocol operations. Fourth, the mechanics of outgoing message sends are handled by a separate hardware unit.

Figure 4.3 shows the structure of the control macropipeline. Message headers are passed from the PI, NI and IO to the *inbox*, which contains the hardware dispatch and speculative memory logic. The inbox passes the message header to the flexible controller — the *protocol processor* (PP) — where the actual protocol processing occurs. To improve performance, PP code and data are cached in the MAGIC instruction cache and MAGIC data cache, respectively. Finally, the *outbox* handles outgoing message sends on behalf of the PP, taking outgoing message headers and forwarding them to the PI, NI, and IO for delivery to the processor, network, and I/O subsystem.

As soon as the inbox completes message preprocessing and passes the message header to the PP, it can begin processing a new message. Similarly, once the PP composes an outgoing message and passes it to the outbox, it can accept a new message from the inbox. Thus, the inbox, PP, and outbox operate independently, increasing message processing throughput by allowing up to three messages to be processed concurrently — hence the name "macropipeline." The following sections describe the operation of the inbox, PP, and outbox in greater detail.

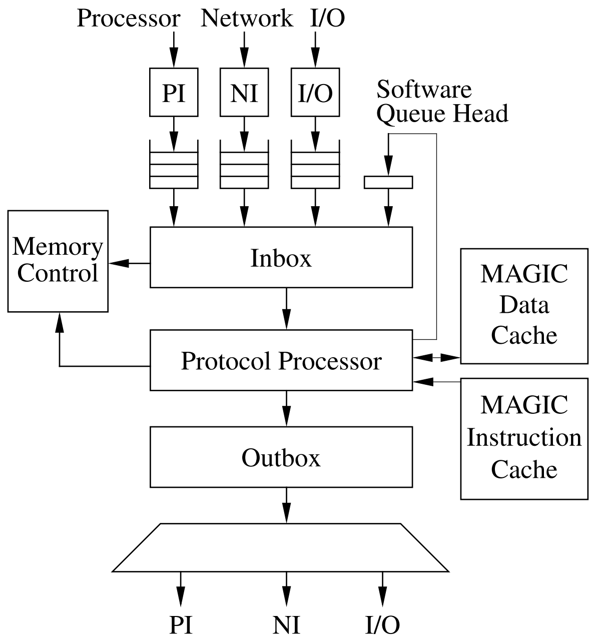


Figure 4.3 Control macropipeline

4.1.3 Inbox Operation

The inbox processes messages in several steps. First, the *scheduler* selects the incoming queue from which the next message will be read. Second, the inbox uses portions of the selected message's header to index into a small memory called the *jump table* to determine the starting PP program counter (PC) appropriate for the message. The jump table also determines whether the inbox should initiate a speculative memory operation. Finally, the inbox passes the selected message header to the PP for processing.

The scheduler selects a message from one of several queues. The PI and IO each provide a single queue of requests issued by the processor and I/O subsystem, respectively. The NI provides one queue for each network virtual lane. The last queue is the software queue. Unlike the other queues, the software queue is managed entirely by the PP. The inbox contains only the software queue's head entry; the remainder of the queue is maintained in main memory, though in many cases it also will be present in the MAGIC data cache.

The scheduler plays a crucial role in the deadlock avoidance strategy. Each incoming queue has an associated array of status bits which the PP can use to indicate whether messages on the queue require outgoing queue space for servicing, or disable scheduling from the queue completely. By initializing these status bits appropriately, the PP can ensure that all handlers can run to completion, that request messages are selected only when sufficient outgoing queue space exists, and that reply messages are selected regardless of the state of the outgoing queues.

By providing programmable, hardware-based message dispatch, the jump table frees the PP from having to perform an inefficient software dispatch table lookup before processing a message. A jump table entry contains a pattern field, a speculative memory operation indicator, and a starting PC value. The inbox compares the pattern field of each jump table entry to the message type, several bits of the address contained in the message, and other information to determine which entry matches the message. The matching entry's speculative memory indicator specifies whether to initiate a speculative read or write for the address contained in the message. The starting PC value specifies the PC at which the PP will begin processing the message. When the PP requests the next message from the inbox, the inbox inserts the starting PC value associated with the new message directly into the PP's PC register. The PP can program the jump table entries to change the particular set of handlers in use.

4.1.4 Protocol Processor Operation

Each protocol requires different message types, state information, state formats, and handlers. To accommodate this variation in processing needs, the PP implements a subset of the MIPS instruction set [57] with extensions to accelerate common protocol operations. These extensions include bitfield extraction and insertion, branch on bit set/clear, and "field" instructions which specify a mask under which the operation is performed. Additional instructions implement the interface between the PP and the other MAGIC functional units, such as issuing message sends to the outbox, requesting new messages from the inbox, and programming the jump table. Thirty-two 64-bit general-purpose registers provide scratch space for use by PP code during protocol processing.

The PP itself is a 64-bit, statically-scheduled, dual-issue superscalar processor core. It fetches an instruction pair from the MAGIC instruction cache each cycle and executes both instructions unconditionally. The PP does not support interrupts, restartable exceptions, or hardware address translation. Although these features can simplify some aspects of protocol processing, they were eliminated to reduce design and implementation complexity. Hence, because the PP lacks many of the resource conflict detection features present in most contemporary microprocessors, the burden of avoiding resource conflicts and generating well-scheduled code falls on the PP programmer or compiler.

4.1.5 Outbox Operation

The outbox performs outgoing message sends on behalf of the PP. It provides a high-level interface to the PI, NI, and IO, relieving the PP from many implementation-specific details such as outgoing queue entry formats and handshaking conventions.

To initiate an outgoing message send, the PP composes the outgoing message header in its general-purpose registers. Next, the PP issues a “message send” instruction. The outbox detects the message send, makes a copy of the message header, inspects the destination of the outgoing message, and passes the message to the PI, NI or IO, as appropriate. Copying the message header requires only one cycle, permitting the PP to proceed almost immediately with additional message processing while the outbox formats the outgoing message and delivers it to the proper interface unit.

4.1.6 Interface Units

In addition to actual message processing, MAGIC must interface to the network, processor, and I/O subsystem. The three interface units — the PI, NI, and IO — implement these interfaces, isolating the rest of the chip from the interface details.

4.2 Detailed Description of MAGIC Functional Units

The MAGIC design was a major undertaking, particularly in a university environment. Although the entire MAGIC design is implemented on a single ASIC, the design is complex and, overall, has about six times more gates than the combined DASH directory controller, reply controller, and the core portions of the memory and IO boards.[2] As a result, understanding how the MAGIC chip operates requires a reasonably detailed discussion of the internal MAGIC implementation.

Whereas Section 4.1 presented an overview of the MAGIC design, the following sections describe each MAGIC functional unit in detail and provide a basis for understanding the components of the design, how they work individually, and how their functionality combines to form a cohesive system. The order of presentation roughly follows the flow of a message through the MAGIC chip.

2. MAGIC contains approximately 750K gates. DASH’s DC and RC boards, plus the core logic of the IO and memory boards, total approximately 125K gates. Both counts are expressed in the same 2-input gate equivalents used to estimate the gate counts for DASH [65]. However, MAGIC’s gate count includes its on-chip memories. Excluding on-chip memory, the MAGIC to DASH gate count ratio is about 2.5:1.

4.3 Message Headers

Common to all parts of the MAGIC chip is the goal of processing messages efficiently. As Section 4.1 notes, messages comprise two sections: message header and (optional) message data. Some parts of the MAGIC chip — in particular, the data transfer logic — process message data. Other parts of the chip — namely, the control macropipeline — process message headers. The functional unit descriptions that follow are more easily understood with the exact format of the messages the MAGIC chip processes in mind. This section introduces the format of the message headers; the data portion of a message is just a series of data words, and its format is self-explanatory.

Incoming messages originate from three sources: another node, in which case the message is delivered over the network; the processor, in which case the message is delivered over the SysAD bus; and the I/O subsystem, in which case the message is delivered over the PCI bus. One of the primary functions of the network, processor, and I/O interface units is to convert external commands into a common internal message format consisting of a *MAGIC message header* optionally followed by any data associated with the message. As noted earlier in the high-level architecture discussion, the message header is stored in the input queues, from which it will be read by the inbox and then passed to the protocol processor. Any data associated with the message flows into a data buffer. The message header must contain sufficient information for the inbox and protocol processor to identify and perform the requested operation and, in general, to send an outgoing message in response.

4.3.1 Message Header Format

A MAGIC message header consists of two doublewords, depicted in Figure 4.4. The first doubleword of the message header contains the following fields:

- The *message type* field (TYPE). This field specifies the operation to be performed. Although shown as a single field in the figure, the message type field is actually subdivided into two parts. The high four bits of the field (bits 11:8) are the protocol identifier bits; these specify the protocol to which the message belongs. The lower eight bits of the field (bits 7:0) specify the particular operation to be performed.
- The *message length* field (LEN). This field specifies the amount of data that accompanies the message. A message may have no data, one doubleword (8 bytes), or a cache line (128 bytes) of data associated with it.
- The unused field at bit positions 15:14 is undefined and is zero-filled.

63 60	59 56	55 55	54 52	51 40	39 28	27 16	15 14	13 12	11 0
PBUF	WBUF	DBL	ENB	MISC	DEST	SRC	Zero	LEN	TYPE
4	4	1	3	12	12	12	2	2	12

MAGIC message header, doubleword 0

63 0
ADDR
64

MAGIC message header, doubleword 1

Figure 4.4 MAGIC message header format

- The *source* field (SRC). This field specifies the number of the node from which the message originated.
- The *destination* field (DEST). This field specifies the number of the node to which the message should be delivered.
- The *miscellaneous* field (MISC). The use of this field intentionally is undefined. Its intended use is as a general mechanism to permit protocol-specific information to accompany a message. Such information should be in addition to the information already contained in the other message header fields, of course. The Flashpoint [73] performance data collection environment, for instance, uses this field.
- The *byte enable* field (ENB). This field specifies the exact data length for doubleword transfers using encodings identical to the ones the R10000 uses on the SysAD bus to specify the size of uncached loads and stores. The byte enable field is used only when the message length field specifies a data length of one doubleword; for message lengths other than one doubleword, the byte enable field is still present in the message header, but is undefined.
- The *double-buffer flag* field (DBL). This field specifies whether the message data is being manipulated as part of a “double buffer” operation. The details of this process are described in Section 4.7.1.
- The *wrap buffer field* (WBUF). This field specifies the number of the data buffer to be used as the wrap buffer in double buffer operations. See Section 4.7.1 for details.
- The *primary buffer field* (PBUF). This field specifies the number of the data buffer associated with the message.

The second doubleword of the MAGIC message header contains a single field: the *address field* (ADDR). This field specifies the address to which the message applies. Since the R10000 supports only 40 physical address bits, the upper 24 bits of the address field (bits 63:40) usually will be unused, as the address field is almost always used to store physical addresses. However, the entire 64-bit field is defined because certain protocols use the address field to exchange 64-bit virtual addresses.

4.3.2 Address Usage

To support a wider range of per-node memory sizes, the MAGIC chip can be configured for several total machine sizes using internal MAGIC configuration registers. From a user's perspective, the total machine size mainly affects the maximum amount of memory that can be supported on each node. At the level of the operating system and within MAGIC, the effects of total machine size are more complicated, as the total machine size affects how the 40-bit physical addresses issued by the R10000 processors are used.

The 40-bit physical address is divided into three fields: node number, address space bits, and node offset. The node number field determines the *home* node for a particular address — that is, the node that physically contains the memory location corresponding to the given address. Address space bits are intended to allow the address to be used at a finer granularity. In particular, the address space bits are part of the search key used to index into the jump table in the inbox (see Section 4.6.1), and thereby support dispatch based not only on whether the address is local or remote, but also on how the address will be used. Finally, the node offset specifies where in the home node's local memory the address is stored.

The size of the node number field and the size of the address space bits field are independent, and are determined by configuration registers within MAGIC. Table 4.1 lists the three supported machine sizes and the corresponding size of the node number field.

Table 4.1 Supported machine sizes

Maximum Machine Size	Size of Node Number Field
256 nodes	8 bits
1024 nodes	10 bits
4096 nodes	12 bits

The number of address space bits can be configured from 0 to 3, independent of the maximum machine size. Given a setting for the machine size and the number of address space bits, the number of node offset bits is simply:

$$\text{(node offset bits)} = 40 - \text{(node number bits)} - \text{(address space bits)}$$

Once the sizes of the three fields have been determined, the location of the three fields within the full 40-bit address is as follows:

- The most significant N bits are the node number bits (N = 8, 10, or 12).
- The next most significant M bits are the address space bits (M = 0, 1, 2, or 3).
- The remaining (40 – N – M) bits are the node offset bits.

This partitioning scheme allows a wide variation in amount of per-node memory and allows a tradeoff between memory size and availability of address space bits. For example, the size and location of the address fields for a 1024-node machine size with two address space bits are illustrated in Figure 4.5.

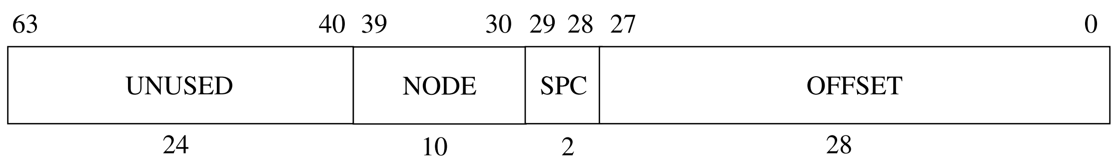


Figure 4.5 Address component example

4.4 Data Transfer Logic

Although the primary function of the MAGIC chip is protocol processing, the need to transfer data efficiently also is central to good performance. The data transfer logic — composed of the data buffers, data buffer allocator, and memory controller — is responsible for storing data while the associated message header is processed by the control macropipeline and for delivering the data as efficiently as possible to the proper destination interface unit once processing is complete. The primary challenge in the design of the data transfer logic is providing both high data transfer bandwidth and low latency. High bandwidth is needed to match the capabilities of the MAGIC external interfaces. The network, memory system, and compute processor can simultaneously deliver data at several hundred megabytes per second; without corresponding bandwidth within MAGIC, the data transfer logic would pose a serious performance bottleneck. Low latency is also important, especially for cache-coherent shared memory in which a processor is often stalled on a cache miss until the data is returned.

Complicating the implementation of the data transfer logic is the need to maintain flexibility. A flexible data transfer mechanism cannot make assumptions about where incoming data will be needed and implement dedicated connections to provide the necessary connectivity. Instead, a general mechanism is needed to accept data from and deliver data to multiple sources simultaneously under the direction of the protocol processor. This separation of policy and mechanism — the protocol processor determines where data will be sent and the data transfer logic handles the actual transfer of the data — is central to achieving flexibility, high bandwidth, and low latency. The sections below further explain the operation of the data transfer logic components.

4.4.1 Data Buffers

Data buffers are used to store any data associated with a message while the message is processed by the inbox, protocol processor, and outbox. Each data buffer is one cache line in size, 128 bytes in the FLASH system. When a message enters the MAGIC chip, the interface unit allocates a data buffer and assigns it to the incoming message, storing the data buffer number in the primary buffer number field of the message header. A data buffer is allocated to every message, regardless of whether the message actually has any associated data when it enters the chip. If the incoming message does contain data, the interface copies the data into the newly-allocated data buffer. Data buffer allocation and other buffer management functionality is discussed in Section 4.4.2, which describes the data buffer allocator unit.

In addition to their data storage function, the data buffers also provide support for data pipelining. That is, they permit the functional unit reading a data buffer (the "sink" unit) to begin the read operation even before the functional unit supplying the data (the "source" unit) has completed its write operation. Data pipelining is achieved by tagging each of the 16 doublewords within a buffer with a *valid bit*. All valid bits are cleared when the buffer is allocated. Thereafter, when the source unit writes a doubleword of data into the buffer, the data buffer logic sets the associated valid bit. The sink unit can then read from the data buffer at any time; however, any read operation that targets a doubleword whose valid bit is clear must be retried.

Each data buffer also includes a single error bit, used to indicate that one or more of the data doublewords in the buffer has been corrupted. The error bit is cleared along with the valid bits when the buffer is allocated. Thereafter, whenever a source unit detects that the data it is writing into the buffer has been corrupted, it sets the buffer's error bit. Sink units can use the state of the error bit to ensure that data corruption information is retained as the data flows through the MAGIC chip and network.

To further improve data pipelining performance, each data buffer is tagged with two status bits, called the *full bit* and the *partially full bit*. Like the valid bits, the two data buffer status bits are cleared when the buffer is allocated. The partially full bit is used as a hint that the buffer contains at least one doubleword of valid data and indicates when a sink unit should begin reading from the buffer. The full bit is not a hint. When set, it indicates that all the doublewords in the buffer contain valid data.

Sixteen data buffers are provided on-chip, each large enough to store one cache line (16 doublewords) of data. The data buffers are implemented as a large multiported RAM array, with a total size of (16 buffers) x (128 bytes/buffer) = 2048 bytes (16384 bits). Multiporting is needed to allow many MAGIC functional units to access the data buffers concurrently. To avoid becoming a performance bottleneck, the data bandwidth achievable through the data buffers must at least match that supported by the rest of the MAGIC chip. To allow simultaneous remote memory reads (data transfer from memory to network) and remote data replies (data transfer from network to processor), a minimum of four data buffer ports are required. The actual implementation includes six ports to provide access for the IO interface unit and protocol processor in addition to the four ports noted in the previous sentence. The design of the data buffer memory array was the work of Ron Ho and Evelina Yeung.

4.4.2 Data Buffer Allocator (DBA)

Because the data buffers are a shared resource, a mechanism is required to allocate new data buffers, to track which buffers are currently in use, and to deallocate buffers when they are no longer needed. The data buffer allocator (DBA) performs all these tasks, managing the allocation and use of the 16 data buffers among the various MAGIC functional units. Like the data buffers themselves, the data buffer allocator is multiported. Each port consists of a set of control lines through which a functional unit can issue commands.

To track the use of the data buffers, the data buffer allocator maintains a *pending counter* for each of the 16 data buffers. The pending counter is simply a small counter that allows the DBA to determine whether a buffer is presently in use by one or more MAGIC functional units. Initially, a data buffer's pending counter is equal to zero, indicating that the buffer is currently unused and hence is available for allocation. When a functional unit requests that a buffer be allocated for its use, the DBA first selects a buffer whose pending counter is currently zero and then completes the allocation request by incrementing the selected buffer's pending counter to one. Thereafter, any MAGIC functional unit that uses the buffer must increment the buffer's pending counter (by issuing a DBA command) before it begins using the buffer, and then decrement the buffer's pending counter when it has finished using the buffer. This series of increment/decrement operations will eventually return the pending counter value to one, regardless of interleaving. To indicate to the DBA that the buffer is no longer in use, however, the pending counter must be decremented one final time to a value of zero. This final decrement is referred to as a buffer deallocation, and is typically performed under the direction of a handler running on the PP. The deallocation can be performed at any time; the combination of the increment/decrement pairs plus the final deallocation decrement will cause the pending counter to return to zero after the last functional unit finishes using the buffer, regardless of the interleaving of the increments, decrements, and the deallocation. Only when the pending counter reaches zero will the DBA once again consider the buffer to be available for allocation.

Although the discussion above appears to imply that all the data buffers are treated as a single, common pool for purposes of allocation, a more sophisticated allocation strategy is required to avoid a potential deadlock condition. This condition is discussed fully in the next section, but for now the key point is that certain functional units require that a data buffer be reserved for their exclusive use. A *reserved buffer* can be allocated only to the functional unit for which it is reserved; other functional units will not be allocated the reserved buffer even if no other buffers are available. Hence, for allocation purposes each of the 16 data buffers is designated either as a general buffer available for allocation to all functional units, or as a reserved buffer available for allocation only to the functional unit for which it is reserved. The assignment of buffers for allocation is controlled by a set of internal DBA registers.

The DBA maintains sixteen pending counters, one for each data buffer. Functional units control the pending counters by issuing commands to the DBA, listed in Table 4.2. No explicit deallocate command exists. Though conceptually distinct from a decrement command, the functionality of a deallocate command is identical, and thus an explicit deallocate command is not required.

4.4.2.1 Deadlock Discussion

Because the NI, PI, and IO interface units must allocate a data buffer to each incoming message, the interface units will stall if no data buffers are available. If the MAGIC chip can enter a state in which all the interface units are stalled waiting for data buffers to become available, but no data buffers will be deallocated, then the MAGIC chip will deadlock. Whether such a deadlock scenario can occur is a function both of the particular protocol(s) in use and of the MAGIC chip's use of the incoming queues. This section will

Table 4.2 Data buffer allocator commands

Command	Function
Allocate	Allocate a buffer
Increment	Increment a buffer's pending counter value
Decrement	Decrement a buffer's pending counter value
Query	Return the current value of a buffer's pending counter
Set	Initialize a buffer's pending counter
Clear	Clear a buffer's valid, partially full, and full bits

describe a deadlock scenario that can occur in a typical cache coherence protocol, though the circumstances that lead to deadlock under this protocol are probably present in most protocols that would be run on the machine.

In a typical cache coherence protocol, the four network incoming queues and four network outgoing queues are divided statically into two types: request queues and reply queues. This division is required to prevent a more basic type of deadlock situation in which request messages and reply messages compete for the same network resources, causing a circular dependency and hence deadlock; this section describes a different type of deadlock situation. Section 4.6.1.1 discusses how the inbox selects messages from the incoming queues, but the key points for this deadlock discussion are:

- The inbox may select a message from an incoming request queue only if each of the outgoing reply queues has free slots.
- The inbox may select a message from an incoming reply queue at any time since reply messages require no outgoing queue space.
- Whenever a message arrives from the network, the NI allocates a data buffer for the message before placing the message in the appropriate incoming queue.

Deadlock can occur under these restrictions if:

- The DBA does not implement reserved buffers, and thus all data buffers are treated as a common pool, available for allocation to any functional unit.
- The PP processes several requests that collectively require enough outgoing reply messages to be sent that all the outgoing reply queues become filled.
- While the PP is processing these requests, more requests arrive. In pulling these requests off the network, the NI allocates all available data buffers.
- The MAGIC chips at the destinations of these outgoing reply messages are in the same situation. Thus, the local outgoing reply queues will not empty because neither the destination nodes nor the local node can allocate data buffers to accept the incoming reply messages.

At this point the inbox cannot select a message from an incoming request queue because of insufficient free space in the outgoing reply queues. To make forward progress, the local and destination inboxes must begin processing reply messages. Processing these replies will relieve the network backup and cause space to open in the outgoing reply queues, permitting the inboxes to once again process requests. Unfortunately, the lack of available data buffers prevents the NIs from accepting any replies from the network, and thus all the incoming reply queues are empty, preventing the inboxes from processing replies. A circular dependency now exists in which data buffers are needed to accept incoming replies, but data buffers will not be made available unless incoming requests can be processed, and incoming requests cannot be processed until incoming replies are accepted.

The mistake in this situation was allowing the NI to allocate all available data buffers to incoming requests. To eliminate this deadlock possibility, the DBA does not treat the data buffers as a single allocation pool. Instead, the buffers are partitioned statically into the following five subgroups:

- One buffer for *each* of the four network incoming queues. These buffers are not treated as a common pool of four buffers; each incoming queue has its own reserved buffer.
- Twelve buffers for general allocation. These buffers are treated as a common pool and are available for allocation to all functional units.

Dividing the buffers into these subgroups ensures that the deadlock scenario described above cannot occur. Although the PP still could fill the outgoing reply queues, the data buffers reserved for the incoming reply queues ensure that additional requests cannot consume all the remaining data buffers. These reserved incoming reply queue buffers also ensure that the NI has at least one data buffer available to accept incoming replies, thus guaranteeing that the PP can free space in the outgoing reply queues by processing incoming replies. At worst the reserved reply buffers are the only free data buffers, in which case the NI pulls a single reply off the network using the buffer reserved for that incoming reply queue, passes the message to the inbox, and then waits for the PP to process the reply and free the reserved buffer. Because the single data buffer serializes the NI, inbox, and PP, overall performance degrades drastically but MAGIC continues to make forward progress.

One might expect that an analogous situation could cause deadlock with the PI or the IO interfaces, but for reasons that go beyond the scope of this discussion, it turns out that not being able to accept new messages from these interfaces does not lead to a deadlock situation. Hence, only the NI incoming queues require reserved buffers. In other protocols, of course, different sets of functional units may require reserved buffers.

4.4.3 Memory Controller (MC)

The memory controller manages the node's local memory system, processes memory read and write transactions from the inbox and PP, and services PP cache fills and write-backs. The node's local memory consists of two memory banks, each containing several memory modules and some external data multiplexor components. To reduce MAGIC pin count requirements, the memory data lines are routed to and from the memory modules via the data muxes. Memory address and control signals bypass the data muxes and are routed directly from the MC to the memory modules. Figure 4.6 illustrates the connections among the MAGIC chip, the data muxes, and the memory modules. The design of the memory controller logic was the work of David Nakahira.

The data muxes serve two purposes: to reduce the MAGIC pin count as noted above, and to perform a frequency conversion between the 100 MHz clock rate used on the port from the MAGIC chip to the data muxes and the 50 MHz clock rate used on the ports from the data muxes to the memory modules. By controlling the two memory banks such that each bank supplies 64 bits of data on alternating 50 MHz cycles, the MC can achieve the full 800 MB/s bandwidth possible on the data mux–MAGIC port.

In addition to managing the bank timing, the MC also provides error detection and correction. Each doubleword in the memory has an associated 8-bit ECC. On memory reads, the MC checks the ECC of each data doubleword returning from memory. Single-bit errors are corrected within the MC and are not visible to other MAGIC units. Double-bit

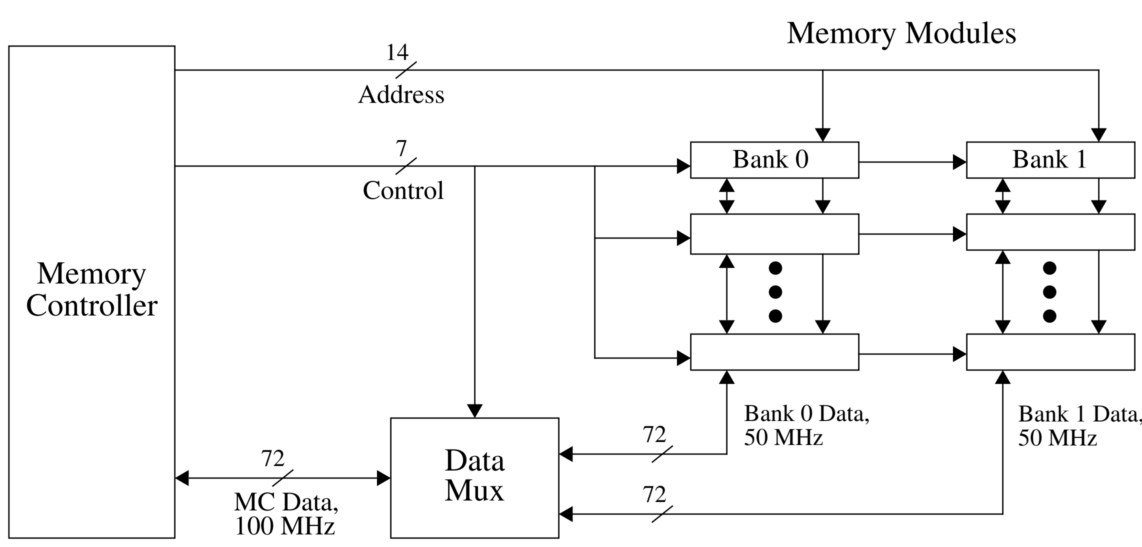


Figure 4.6 Memory system overview

and other uncorrectable errors cause the MC to set the error bit for the data buffer (see Section 4.4.1). On memory writes, the MC inspects each outgoing data doubleword and generates the proper ECC, which is written to the memory modules along with the data.

The memory controller supports both single-buffer, cache line-sized reads and writes, as well as doubleword reads and writes. The controller also supports double-buffer reads, which are used to accelerate unaligned message passing operations, as discussed in Section 4.7.1. For single-buffer, cache line-sized operations, the address must be doubleword-aligned and the MC transfers data in *subblock order*, as defined in the R10000 specification. Thus on memory reads, the MC writes data doublewords into the data buffer in subblock order, and on memory writes the MC expects the data to be stored in the data buffer in subblock order. The PP caches also are configured to accept fill data and supply writeback data in subblock order. Figure 4.7 illustrates the use of subblock ordering for an example cache line-sized memory read.

Example: Single-buffer memory read from byte address 0x2FC8

Memory Byte Address

2FC8	2FC0	2FD8	2FD0	2FE8	2FE0	2FF8	2FF0	2F88	2F80	2F98	2F90	2FA8	2FA0	2FB8	2FB0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Data Buffer Doubleword Offset

Figure 4.7 Example single-buffer, subblock-ordered memory read operation

4.5 Interface Units

The data buffers, data buffer allocator, and memory controller form the core of the MAGIC data transfer logic. The remaining portions of the chip are interface units and the control macropipeline. The interface units, as described in Section 4.1, include the network interface, processor interface, and I/O interface. These units not only provide the physical interfaces to the network, compute processor, and I/O subsystem, respectively, but also handle the splitting and recombination of message headers and message data. In addition, the interface units perform data format conversions between the external world and the internal MAGIC message format, and perform certain preprocessing tasks in preparation for passing message headers to the inbox and protocol processor.

4.5.1 Network Interface (NI)

The NI performs two major operations: forwarding outgoing messages from the NI output queues to the network, and forwarding incoming messages from the network to the NI incoming queues. In addition to isolating the rest of the MAGIC design from the details of the MAGIC–router interface, the NI also performs message routing calculations, message error detection, and precomputes some of the values that are used later to index into the jump table in the inbox.

The Spider router component that implements FLASH's interconnection fabric supports four virtual lanes across each of the two physical links (incoming and outgoing) that connect one router to another router, or a router to a MAGIC chip. Accordingly, the NI also supports four incoming and four outgoing virtual lanes. It provides four output queues and four input queues, one for each virtual lane and, like the router, allows all virtual lanes to operate independently. Logically, the NI consists of two subunits: the NI incoming logic, which implements the router to NI input queues path, and the NI outgoing logic, which implements the NI output queues to router path.

4.5.1.1 Network Message Format

Within the MAGIC chip, messages consist of a MAGIC message header, as shown in Figure 4.4, and any associated message data. When messages are transferred over the network, their format must be altered to accommodate additional routing and other network control information. As described in Section 3.3, the network breaks messages into 128-bit packets called micropackets. Thus, to translate a message from the internal MAGIC format to the network format, the NI outgoing logic must add additional routing and control information and break the entire message into a series of micropackets. The NI incoming logic at the destination node reverses this process, taking a series of micropackets and restoring the internal MAGIC message header and data format.

All messages on the network begin with one network header micropacket. The format of this micropacket is shown in Figure 4.8 and is dictated by the design of the Spider chip, whose internal logic assumes that all messages start with a header micropacket in the format shown. The network header is a mixture of the MAGIC message header fields from Figure 4.4 and three additional fields that are interpreted by the router. These additional fields are the message age (AGE), direction (DIR), and congestion control (CC), which, along with the destination field, control message routing and priority through the network. The asterisks in the figure indicate that these fields are the only portions of the network header that the router interprets. The router ignores other fields within the network header, which pass unchanged from source to destination.

The destination field is nine bits in the network header but twelve bits in the MAGIC header. The Spider router supports only a 9-bit destination field, so the upper three bits of the 12-bit MAGIC destination are discarded as the MAGIC header is translated into a network header. As a result, the maximum FLASH system size with the current Spider router is 512 nodes even though the MAGIC chip itself can support up to 4096 nodes.

63 61	60 49	48 37	36 35	34 23	22 14	13 10	9 2	1 0
ENB	MISC	SRC	LEN	TYPE	DEST*	DIR*	AGE*	CC*
3	12	12	2	12	9	4	8	2

Network message header, doubleword 0

63 0
ADDR
64

Network message header, doubleword 1

Figure 4.8 Network header micropacket format

For a message that contains no data, the entire message consists solely of the network header. For messages with data, the data is placed in additional micropackets that are transferred after the network header. In addition to messages without data, MAGIC supports two data lengths: one doubleword of data or one cache line of data. The network formats of these messages are shown in Table 4.3 and Table 4.4.

Table 4.3 Network message format, one doubleword of data

Micropacket	Contents
0	Network header
1	Data doubleword in bits [63:0]; Bits [127:64] undefined

Table 4.4 Network message format, one cache line of data

Micropacket	Contents
0	Network header
1 – 8	Data doublewords 0 – 15; two doublewords per micropacket

Table 4.5 summarizes the total message size (in micropackets) and the associated peak network data bandwidth achievable with the specified network message formats, assuming a peak raw network transfer rate of 800 MB/s. Data bandwidth refers to the number of bytes of actual data that can be transferred per unit time, and thus excludes the network header micropacket. It is calculated by dividing the actual amount of data in a message by the total message size and then multiplying by 800 MB/s.

Table 4.5 Total message size and data bandwidth

Message Length	Total Message Size	Data Bandwidth
No data	1 micropacket	N/A
One doubleword	2 micropackets	200 MB/s
One cache line	9 micropackets	711 MB/s

4.5.1.2 NI Incoming Logic

Figure 4.9 depicts an overview of the NI incoming logic path. Micropackets arriving from the network are received into the incoming LLP module, which manages the link-level aspects of the network protocol. Specifically, the incoming LLP receives micropackets from the router, checks the micropacket CRC to see if the micropacket was corrupted in transit, and requests retransmission if an error occurred. The incoming LLP also manages flow control between the MAGIC chip and the router, indicating whether additional micropackets can be accepted on each virtual lane.

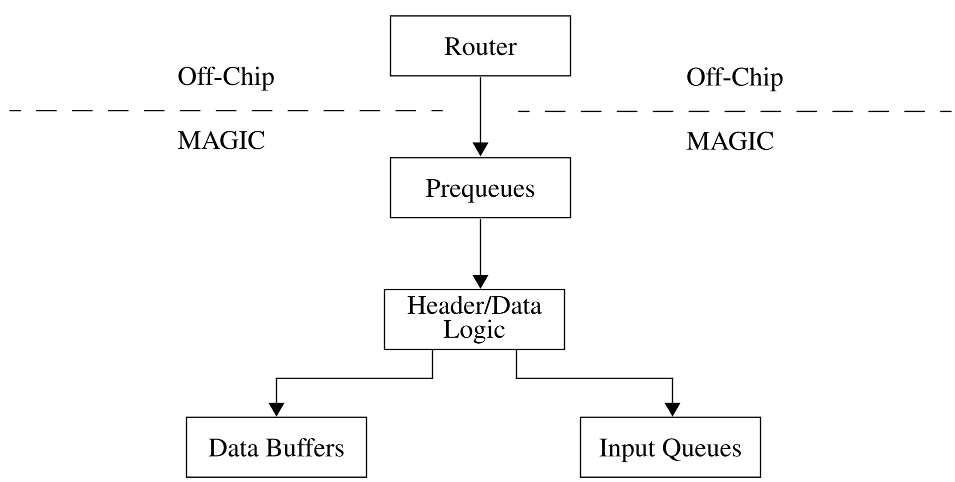


Figure 4.9 NI incoming logic overview

From the incoming LLP, micropackets enter the prequeues module where they are queued in one of the four prequeues, depending on the virtual lane on which the micropacket arrived. The NI incoming header/data logic then reads the micropackets from the prequeues, allocates a data buffer for the incoming messages, converts the network header into a MAGIC message header, and stores the header in the proper NI input queue. It also precomputes certain message characteristics that are later used to index into the jump table in the inbox; see Section 4.6.1.2 for details. The message data micropackets flow from the prequeues into the header/data logic and then to the proper data buffer. In this way, the header/data logic splits the message into header information (stored in the NI input queues) and data (stored in the data buffers).

4.5.1.3 NI Outgoing Logic

Figure 4.10 shows an overview of the NI outgoing logic. The outbox stores outgoing message headers into one of four output queues, according to the virtual lane on which the PP sent the outgoing message. Like the NI incoming logic, the NI outgoing logic consists mainly of a header/data logic block. This logic reads outgoing message header from the output queues, converts the header into the network header format shown in Figure 4.8, computes the additional routing and control fields needed by the router, and forwards the network header micropacket to the outgoing LLP module. Then, if the message contains data, the data is read from the appropriate data buffer, split into micropackets, and forwarded to the outgoing LLP. The outgoing LLP logic takes the outgoing micropackets, adds CRC information where required, and transmits the micropackets to the router.

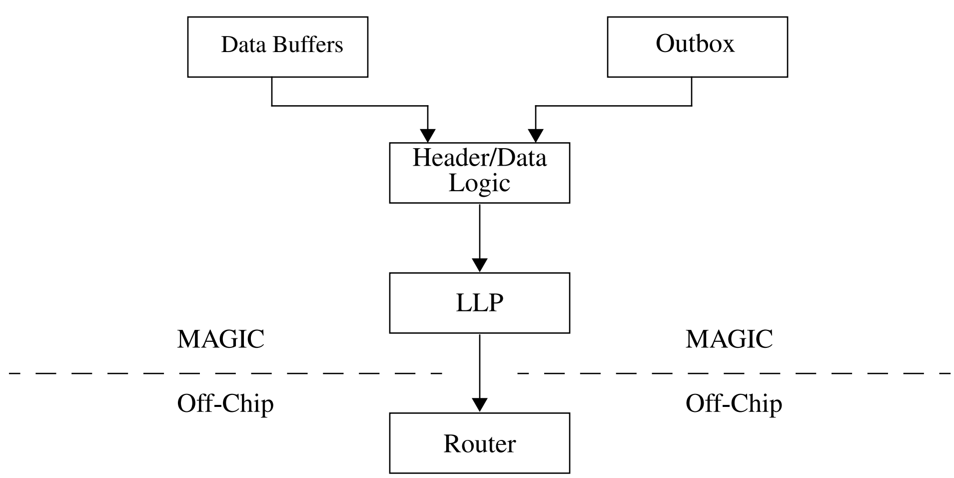


Figure 4.10 NI outgoing logic overview

4.5.2 Processor Interface (PI)

The processor interface handles communication with the R10000 compute processor via the SysAD bus. The PI handles incoming messages originating from the R10000 and outgoing messages arriving from the outbox. However, because the SysAD bus itself and many of the internal PI resources are shared, the PI is best viewed as three submodules: one to process incoming messages from the R10000, one to process outgoing messages arriving from the outbox, and a third submodule to handle shared resource arbitration and allocation between the first two submodules. Figure 4.11 depicts the internal organization of the PI. The design of the PI logic was the work of David Ofelt.

To a much greater extent than the NI, the PI bases its operation on the contents of several outgoing message header fields. For this reason, the PP is not free to define the message type encodings used in messages sent to the R10000; instead, these encodings must match those used on the SysAD bus, regardless of the protocol(s) in use in the system.

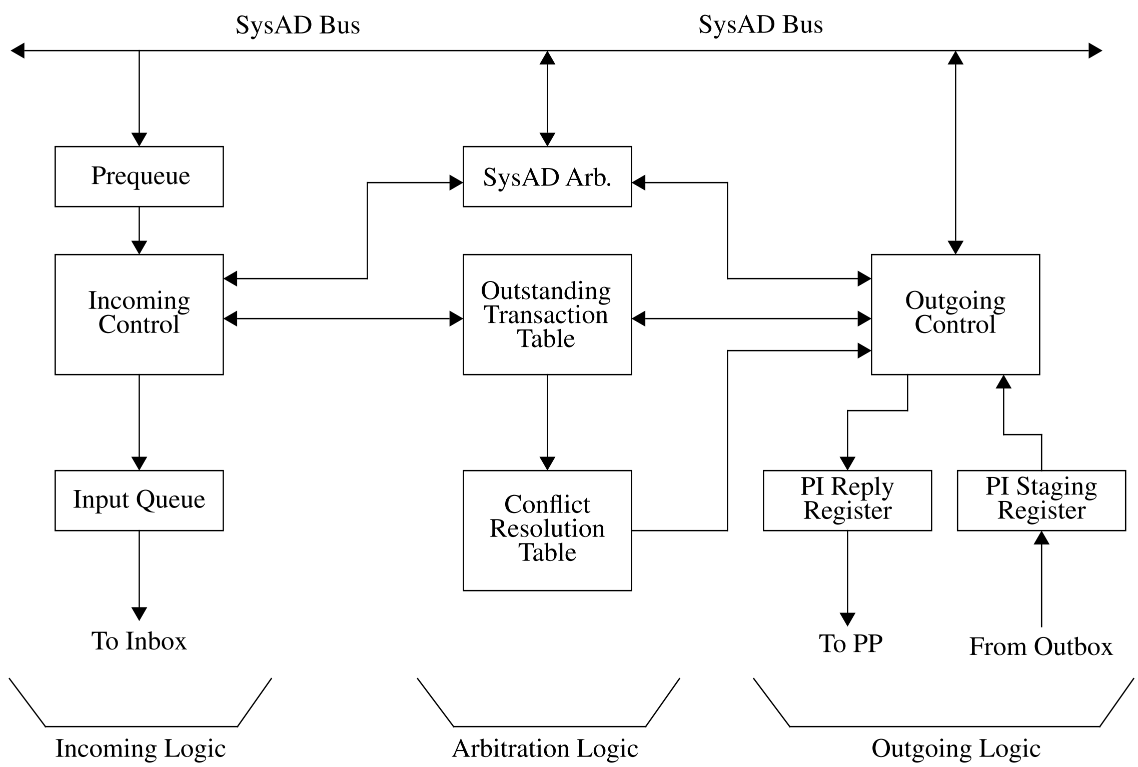


Figure 4.11 PI overview

4.5.2.1 Arbitration, Conflict Detection, and Conflict Resolution

One function of the PI arbitration logic is to manage SysAD bus arbitration and flow control. The incoming logic, outgoing logic, and R10000 all can request use of the SysAD bus. The SysAD bus arbiter grants use of the bus to these modules on a transaction-by-transaction basis. Because the SysAD bus specification requires a dead cycle between bus ownership transfers, a strict round-robin arbitration policy could impact overall SysAD utilization by generating an unnecessarily large number of ownership transfers. For this reason, the SysAD arbiter attempts to optimize SysAD utilization by granting the bus to the same requester so long as that requester has another transaction ready and so long as all deadlock possibilities are avoided. The SysAD arbiter also must manage SysAD flow control. When the incoming logic cannot accept new read transactions or write transactions, it notifies the SysAD arbiter, which controls the proper SysAD signals as necessary.

Another function of the arbitration logic is to track all outstanding transactions[3] and manage conflicts that occur between an outstanding transaction and an outgoing message (that is, a message from the PP to the R10000). A conflict occurs whenever the address contained in an outgoing message matches the address of an outstanding R10000 transaction. Conflict detection requires that the address contained in an outgoing message be compared to the addresses of all outstanding transactions to determine whether a conflict has occurred. Conflict resolution requires that the command contained in the outgoing message and the command contained in the conflicting outstanding transaction be inspected to determine the proper conflict resolution procedure. In most cases, the presence of a conflict can be ignored. In other cases, however, proper resolution of the conflict requires the outgoing message to be altered or even aborted.

The *outstanding transaction table* (OTT) and *conflict resolution table* (CRT) are used for conflict detection and conflict resolution, respectively. The OTT is a small, fully-associative array, each entry of which contains the address, command, and SysAD bus ID of an outstanding R10000 transaction. Each OTT entry also contains a timeout counter. The CRT is also a memory array, though not an associative one, indexed by a combination of the contents of an OTT entry and a command from the PI outgoing logic.

When the R10000 issues a new transaction on the SysAD bus, the incoming logic inserts the associated address, command, and SysAD bus ID into the OTT. As outgoing messages are received by the PI outgoing logic, the address of the outgoing message is used to search the OTT. If an OTT entry with the same address is found, the command

---

3. A transaction is outstanding if it has been accepted by the PI incoming logic but no completion response has been issued to the R10000 from the PI outgoing logic.

field of the matching OTT entry and the command contained in the outgoing message are concatenated and used to index into the CRT. The output from the CRT controls how the outgoing logic will process the outgoing message. The CRT output specifies that the outgoing message be passed unaltered to the R10000, that the outgoing message be passed to the R10000 but with the command in the outgoing message replaced by a negative acknowledgment, or that the outgoing message be aborted (that is, not delivered to the R10000 at all) and a negative acknowledgment returned to the PP instead. The CRT output also specifies whether the outgoing message is the completion of the matching outstanding transaction. If so, the CRT specifies that the matching OTT entry be deallocated, indicating that the outstanding transaction is now complete.

Table 4.6 lists two example conflict cases and their resolutions. In general, many of the conflict cases can occur only as a result of cache coherence operations, though some conflicts are likely to be present in any protocol.

Table 4.6 Example conflict cases

Conflict Description	Resolution Description
A read reply is received for a line that has been invalidated while the read transaction was outstanding.	Send a negative acknowledgment to the R10000 instead of the read reply data.
An exclusive intervention request is received for a line that has an upgrade request outstanding.	Abort the outgoing message and return a negative acknowledgment to the PP.

4.5.2.2 Incoming Message Processing

Incoming messages, in the form of SysAD bus transactions, originate from the R10000 and flow into the incoming control logic. To improve performance, the incoming logic attempts to preallocate a data buffer so that new incoming messages can be accepted and processed immediately, without first having to allocate a data buffer at the time the message arrives. If no data buffers are available, or if the incoming queue is full, the message is stored temporarily in the prequeue and the SysAD arbiter is notified so that it can manipulate the SysAD arbitration process and prevent the R10000 from issuing new transactions. Assuming that a data buffer has been preallocated, the incoming logic reads the SysAD transaction, converts it into a MAGIC message header, and stores the header in the PI input queue. Any data associated with the message is stored in the data buffer that has been allocated. In addition to allocating a data buffer and transferring the message header and message data, the incoming logic also updates the OTT as noted above.

4.5.2.3 Outgoing Message Processing

Unlike incoming messages, outgoing messages are not stored in a queue. Instead, the PI implements a single outgoing message staging register into which the outbox writes outgoing message headers. Once an outgoing message has been written into the staging register, the PI will not accept a new outgoing message until it completes processing the current one. The outgoing message header is passed through the OTT and CRT as described in Section 4.5.2.1, converted into the proper format for a SysAD transaction, and then issued on the SysAD bus (unless the CRT indicates that the outgoing message should be aborted). If the message type indicates that the message has associated data, the outgoing logic transfers the data from the data buffer to the SysAD bus.

For some outgoing message types, the R10000 will return a response to the PI. This response consists of a state indication and, for certain message types, data retrieved from the R10000's cache. The state portion of the response is stored in the PI reply register, from which it can be read by the PP. Any data that accompanies the response is stored in the data buffer that was specified in the outgoing message header.

4.5.3 I/O Interface (IO)

The IO interface unit implements the FLASH node's local PCI-based I/O subsystem [81], forming the interface between the MAGIC chip and all devices attached to the local PCI bus. The design of the IO logic was the work of Hema Kapadia.

Providing efficient I/O service is challenging in distributed multiprocessors for several reasons. First, I/O devices are characterized by a large variation in data bandwidths, with most devices providing significantly lower bandwidth than the other MAGIC interfaces. Data latency also is typically larger on I/O devices. Consequently, the design of the I/O interface unit must not only accommodate these varying bandwidth and latency characteristics, but do so in a manner that does not adversely impact the performance of the other MAGIC interfaces.

Second, the PCI bus itself is not well-suited for use in a multiprocessor. With a bus protocol targeted and optimized for personal computers, the PCI bus does not mesh efficiently with a multiprocessor memory system. Assumptions about the ability of the host memory system to provide transaction completion information and assumptions about the ability to complete a transaction once it is issued on the bus work well in uniprocessors or even bus-based multiprocessors, but are unsuited for the variable latency and general lack of atomicity of a distributed multiprocessor memory system. Overall, then, the IO interface unit must not only provide the necessary logical interface and translation mechanisms to allow the MAGIC chip to communicate with the attached PCI devices, but also must layer additional functionality on the interface logic to adapt the PCI bus characteristics to a distributed multiprocessor environment.

All of these requirements are managed in the IO interface unit. The block diagram is shown in Figure 4.12. Like the processor interface, the IO interface is best viewed as three submodules: a PCI bus arbiter and bus interface unit, a unit to handle outgoing messages from the protocol processor to a PCI device, and a unit to handle incoming messages from a PCI device.

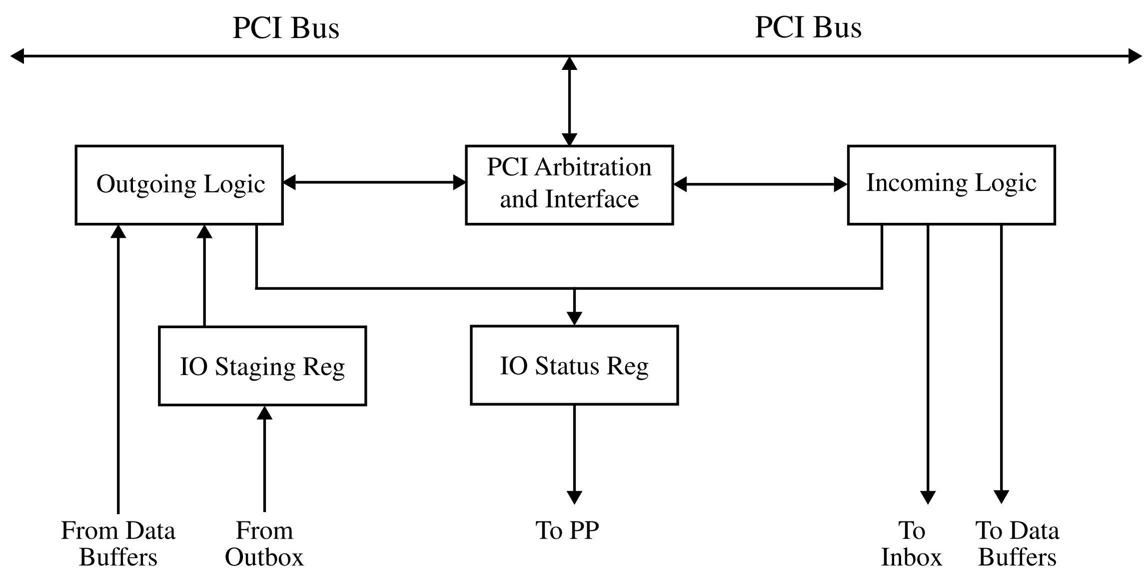


Figure 4.12 IO overview

Outgoing messages are translated into programmed I/O (PIO) transactions on the PCI bus. Incoming messages — those originating from a PCI device — are called DMA transactions and are translated into MAGIC messages for processing by the protocol processor. Section 4.5.3.2 below describes PIO and DMA operations in more detail.

4.5.3.1 Coherence

One of the major challenges in integrating an I/O subsystem into a multiprocessor is maintaining coherence between data held in the memory system (that is, in the memories themselves and in the R10000 caches) and data held in the I/O subsystem. The MAGIC IO interface solves this problem by implementing a one-line I/O cache called the collection buffer. The collection buffer actually solves two problems. First, as noted, it provides a means to maintain coherence. Like the compute processor cache, the collection buffer maintains coherence state information and can respond to intervention requests and supply data as needed. Thus the system, typically with a few augmentations to the directory format and cache coherence protocol, can treat each node's I/O system almost as if it were another processor.

Second, the collection buffer, as its name implies, provides buffering for I/O transactions that are smaller than a cache line. Without such buffering, each of these smaller transactions — often just a byte or word in size — would need to be issued into the memory system, causing a substantial performance loss for the I/O operations themselves and interfering with other, non-I/O messages.

4.5.3.2 PIO and DMA Operations

The IO interface supports two transactions: PIOs and DMAs. PIO reads and writes are initiated by the protocol processor, typically in response to an uncached operation issued by a compute processor. The PIO is received from the outbox into the IO staging register, converted into the proper format for the PCI bus by the outgoing logic block, and then issued to the appropriate PCI device. If the PIO requires data transfer, the data is read or written from the data buffer contained in the outgoing message header.

DMA read and write transactions are similar to PIOs except that they are initiated by a PCI device rather than by the protocol processor. The DMA request is received from the PCI bus into the incoming logic block, where it is converted into a MAGIC message header and passed to the inbox for processing. A data buffer is also allocated and used to store any data associated with the DMA.

Complications arise in the servicing of both PIOs and DMAs in cases in which the transaction cannot be satisfied immediately. For example, a DMA read of remote memory can require several hundred cycles to retrieve the requested data. In normal PCI operation, the DMA occupies the bus for the duration of the transfer, preventing other devices and the protocol processor from using the bus for other transactions. Although this convention works well in a uniprocessor or bus-based multiprocessor, it is ill-suited to a distributed multiprocessor. As a result, the IO interface logic provides several mechanisms to abnormally terminate a transaction so that other transactions can be issued on the PCI bus. This capability is necessary to avoid deadlock in the I/O system.

4.6 Control Macropipeline

The interface units and data transfer logic provide an efficient mechanism to receive messages into MAGIC, split the messages into headers and data, store the message data in a data buffer, and deliver the headers to the control macropipeline. How these headers are processed is the realm of the control macropipeline. The inbox, protocol processor, and outbox — the components of the control macropipeline — combine to determine in what order the message headers will be serviced, what protocol actions are needed for each header, what outgoing messages need to be sent in response, and to where the data transfer logic should deliver any data associated with the message. Close coupling among the inbox, protocol processor, and outbox is central to performing these tasks efficiently.

4.6.1 Inbox

The inbox is responsible for selecting the next message to be processed by the PP and for performing certain message preprocessing activities, including message type dispatch and speculative memory operation initiation. By performing these repetitive preprocessing tasks, the inbox increases the efficiency of the MAGIC macropipeline by freeing the PP from performing these tasks and by allowing the preprocessing to occur while the PP and outbox are working on previous messages. The design of the inbox logic was the work of Mark Heinrich.

Message preprocessing consists of several steps. First the inbox *scheduler* must select the incoming queue from which it will read the next message header to be processed. After selecting a queue, the inbox reads the two MAGIC message header doublewords from the queue and stores them temporarily. At the same time the message header is read from the incoming queue, portions of the message header are used to index into the *jump table*, a memory array that allows for efficient, hardware-based message dispatch and speculative memory operation initiation. One output of the jump table indicates whether the inbox should initiate a speculative memory operation for the address contained in the message header. If this output is asserted, the inbox takes the address and buffer number from the message header along with a read/write indication from the jump table and issues the appropriate memory transaction to the memory controller. Another jump table output contains the starting PC value at which the PP should begin processing the message.

The only exception to this procedure occurs when the scheduler selects the internal *software queue registers* as the source of the next message header to be processed. These registers are maintained entirely by the PP and support temporary suspension of message processing while the PP performs another task. Section 4.6.2.2 discusses the software queue in more detail. The software queue registers contain a MAGIC message header and a starting PC value for the PP. When the software queue registers are selected, the inbox does not use the jump table. Instead, the contents of the software queue registers directly specify the PP's starting PC value, and no speculative memory operation can be initiated.

Once a message header and its associated PC value have been generated, the PP may retrieve the new header from the inbox. The PP issues a *switch* instruction to retrieve the first doubleword of the message header and the PC value. The header is placed in the PP's general-purpose register that is specified as the destination of the switch instruction, and the PC value is injected directly into the PP's current PC register, just as if the PP had performed a MIPS "jump register" instruction. The PP then must issue a *ldctxt* instruction to retrieve the second message header doubleword, which also is placed in one of the PP's general-purpose registers. Upon completion of the switch/ldctxt instruction sequence, the PP has retrieved the entire MAGIC message header of the next message to be processed, has jumped to the proper starting PC, and can begin actual message processing.

4.6.1.1 Inbox Scheduler Operation

The scheduler is responsible for selecting the next message to be processed from the six incoming queues — the PI, IO, and each of the four network lanes feed into separate incoming queues — and the software queue registers. While conceptually straightforward, the actual scheduling process is complicated by the need to avoid inducing deadlock by scheduling a queue when its resource requirements cannot be met. In the absence of deadlock issues, scheduling is a round-robin process, meaning that each incoming queue has equal priority.

Deadlock can occur if a queue is scheduled when insufficient outgoing network queue space exists. In a traditional request–reply protocol, for instance, a request must not be scheduled unless sufficient space exists in the outgoing network reply queue. For this reason, the scheduler considers not only round-robin priority when choosing the next incoming queue, but also the free space available in each of the four outgoing network queues. Several inbox configuration registers specify the outgoing queue space requirements for each of the incoming queues; this information is combined with the actual amount of free space in each outgoing network queue to determine if deadlock considerations require an otherwise valid incoming queue to be temporarily disregarded from scheduling. After deadlock considerations have been taken into account, the round-robin priority scheme governs which incoming queue is selected.

4.6.1.2 Jump Table Operation

Once an incoming queue has been selected in the scheduling process, the two MAGIC message header doublewords are read from the front of the selected queue. Parts of the header are extracted and used to form a 17-bit search key for the jump table. To determine whether a jump table entry matches the jump table search key, the search key is partitioned into two fields: a 6-bit index field and an 11-bit comparison field. The components of the index and comparison fields are noted in Table 4.7 and Table 4.8.

Table 4.7 Jump table index field components

Component Name	Width	Description
Address local/remote	1	Set if the node number field of the address contained in the message header is equal to the local node number
Queue source	1	Set if message originated from NI; clear if message originated from PI or IO
Message type low	4	Bits [3:0] of the message type field

Table 4.8 Jump table comparison field components

Component Name	Bits	Description
Address range	1	Set if the offset portion of the address is outside the memory physically present on the node
Reserved	1	Set if the data buffer associated with the message is the queue's reserved data buffer
Cell	1	Set if the source field of the message header indicates that the message originated from within the local node's fault-containment cell
Address space	3	The three address space bits
Queue source	1	Set if message originated from IO; clear if message originated from PI or NI
Message type high	4	Bits [7:4] of the message type field

The jump table is a 256-entry, four-way associative RAM array, and the index field is used as an address to fetch four jump table entries. Each jump table entry contains, in addition to the starting PC and speculative memory operation indicator, an 11-bit *mask field* and an 11-bit *match field*. The jump table entry matches the jump table key if and only if the following condition holds for each of the 11 bits in the comparison field, mask, and match values:

$$\text{entryMatches} = (\text{comparison} \ \& \ \text{mask}) == \text{match}$$

The structure of this test leads to distinct functionality for each of the four combinations of the mask and match bits, as shown in Table 4.9.

Table 4.9 Effect of mask and match

Mask	Match	Effect
0	0	Corresponding bit of comparison field will always match
0	1	Corresponding bit of comparison field will never match
1	0	Corresponding bit of comparison field will match only if it is equal to 0
1	1	Corresponding bit of comparison field will match only if it is equal to 1

All four jump table entries are searched using this test, and at most one entry is permitted to match the search key; it is an error if the jump table is initialized such that more than one entry can match a given search key. Once a matching entry is found, the speculative memory operation indication and starting PC values are extracted and used to govern further inbox operation as described earlier.

The jump table also contains support for sampling-based performance monitoring. Discussion of this topic is deferred to Section 4.7.2, which describes the sampling support in the jump table as well as other MAGIC performance monitoring facilities.

4.6.1.3 Special PC Values

In normal operation, all message headers — with the exception of those read from the software queue — are passed through the jump table, and the matching jump table entry is used to control the remainder of the inbox's operation for the message. However, several events can alter this flow. First, the jump table may not contain a matching entry. Because the jump table is of limited size, only those entries corresponding to the most common message types can be stored. Receipt of a less common message type sometimes will not locate a matching entry in the jump table, a situation referred to as a *jump table miss*. Second, for messages arriving from the network via the NI, the incoming message header may have been corrupted in transit. The NI flags erroneous message headers by setting a header error bit. Similarly, the NI flags messages that arrive with invalid protocol ID bits in the message type field (see Figure 4.4) by setting a bad protocol ID bit. A separate PC value is dispatched when any of these situations occurs, allowing the PP to process the exceptional condition appropriately.

4.6.2 Protocol Processor (PP)

The protocol processor (PP) is the heart of the control macropipeline and of the MAGIC chip as a whole. Indeed, the main purpose of all other logic in the chip is to support the PP's interface to external components and to improve PP protocol processing performance. The PP provides a programmable mechanism for specifying the actions that should be taken by the memory system in response to incoming messages. Toward this end, the PP is a fully-programmable, dual-issue superscalar microprocessor that accesses instructions and data in the standard physical address space of the FLASH node where it resides. However, due to its intended purpose, the PP departs from existing microprocessor architectures in a number of areas.

Following a brief instruction set summary, this section focuses on the functionality that efficiently couples the PP to the surrounding MAGIC hardware units and improves overall protocol processing performance. The design of the PP logic was the work of Jules Bergmann.

4.6.2.1 PP Instruction Set Architecture Summary

The PP instruction set architecture is a 64-bit load/store architecture loosely based on the MIPS instruction set, with changes incorporated to support the PP's primary function of handling messages efficiently. Like the MIPS ISA, all instructions are 32 bits in length, there are 32 general-purpose registers (GPRs), and byte addresses are used. However, the PP provides no address translation mechanism; all addresses generated by the PP are physical addresses. In addition, the PP does not support interrupts or restartable exceptions. Finally, though the PP is dual-issue, it is entirely statically scheduled and various scheduling restrictions must be observed by the programmer and/or compiler system.

The PP instruction set includes a typical complement of ALU, ALU immediate, shift, branch/jump, and load/store instructions. In addition, it includes support for setting and testing single bits, as well as manipulating bit fields and bit vectors, all of which occur frequently in the data structures used by handler code. Several special instructions support the interface of the PP to the other MAGIC hardware units; these instructions are described in the following subsections. These additions to the instruction set are an important part of providing close coupling between the PP and the rest of the MAGIC chip.

4.6.2.2 Interface to the Inbox

The primary function of the inbox is to prepare messages for processing by the PP. When the PP has finished processing a message, it fetches the next message from the inbox by issuing two special instructions in sequence: a *switch* instruction followed by a *ldctxt* (load context) instruction. The *switch* instruction transfers execution to the appropriate handler for the message by loading the PP program counter with a value supplied by the inbox. The information needed by the handler code to process the message is contained in a header supplied by the inbox that consists of two doublewords. In addition to loading the program counter with a new value, the *switch* instruction also loads the first doubleword of the header into one of the PP's GPRs. The *ldctxt* instruction loads the second doubleword of the header into a GPR. If the PP executes a *switch* instruction when no further messages are available to process, the inbox stalls the PP until a message arrives. The inbox allows the PP to continue execution once the inbox is prepared to complete the *switch* successfully.

In certain cases, the PP may want to suspend its processing of a message temporarily and resume the processing at a later time. The inbox supports this capability through the software queue registers as described in Section 4.6.1.1. This support allows the PP to create and manage a separate inbox queue in memory that is scheduled along with the other hardware queues in the inbox. When the PP wishes to suspend processing of a message, it saves any necessary state information in the software queue registers and memory and indicates to the inbox that the software queue is valid.

4.6.2.3 Interface to the Outbox

The purpose of the outbox is to transfer outgoing messages generated by the PP to their appropriate destination interface unit (i.e., the PI, NI, or IO). The PP transfers a message to the outbox using the *send* instruction. The *send* instruction encoding specifies two source GPRs that contain the two message header doublewords; these doublewords are transferred to the outbox. The encoding also specifies five *message send type bits* that notify the outbox of the message characteristics, such as whether the message contains data in addition to the header.

4.6.2.4 Interface to the Memory Controller

Along with the inbox, the PP also can initiate memory operations. Two sources of PP-initiated memory operations are the PP instruction and data caches, which require memory operations to service a cache miss. A third source of memory operations is PP handlers. In general, the speculative memory initiation capability in the inbox is designed to alleviate the need for the PP to perform its own memory operations, both to reduce PP occupancy and to improve the latency to service a message. In certain circumstances, however, a PP handler will need to initiate memory operations itself. Two instructions are provided for this purpose: *lblock* and *sblock*. The *lblock* instruction initiates a memory read from a specified address into a specified data buffer; *sblock* does the reverse, initiating a memory write from a specified data buffer to a specified address.

4.6.2.5 Interface to MAGIC State: The Misc Bus

To maximize flexibility, testability, and performance monitoring capabilities, the PP must be able to access as much state within MAGIC as possible. With a few exceptions, state access is accomplished through the *ls* (load state) and *ss* (store state) instructions. Each piece of accessible state in MAGIC is assigned a *state address*; this address is specified by the *ls* and *ss* instructions much as a memory address is specified by a regular load/store instruction. These instructions transfer state addresses and data over the *misc bus*, an internal bus that runs to each functional unit within MAGIC for this purpose. By issuing *ls* and *ss* instructions, the PP can program the internal registers in various functional units, probe internal functional unit state for debugging purposes, and perform tasks such as issuing DBA commands and modifying the software queue registers. Table 4.10 summarizes some of the information that is accessible and operations that can be performed via the misc bus.

Table 4.10 Misc bus functionality

Unit	Example Misc Bus Operations
NI	Modify internal registers; read input and output queues
PI	Modify internal registers; read input queue; read PI reply register
Inbox	Modify jump table, software queue registers
Outbox	Modify internal registers
DBA	Issue DBA commands; modify internal registers
Data buffers	Read/write data buffers; modify data buffer status bits
MC	Modify internal registers

4.6.2.6 MAGIC Instruction and Data Cache Operation

All PP instruction fetches go through the MAGIC instruction cache (MIC); all PP loads and stores go through the MAGIC data cache (MDC). Since the PP has no notion of virtual memory, both caches are physically indexed, physically tagged, and backed directly by the node's local memory. The MIC is 16 KB in size and is implemented on the MAGIC die. It supports single-cycle access. The MDC is off-chip and has a total size of 1 MB. Because of the additional delay of an off-chip operation, MDC accesses require two cycles. As a result, load instructions incur two delay slots. Both caches are direct-mapped and use 128-byte lines to maximize compatibility with the MAGIC hardware mechanisms in place to service the 128-byte transfers required for the R10000.

The PP stalls whenever an instruction fetch misses in the MIC or when a load or store misses in the MDC. A hardware sequencer implements cache fills by controlling a direct connection between the PP caches and the memory controller, allowing cache data to be transferred between the caches and the memory system without the need for intermediate storage in a data buffer. To minimize MDC cache miss service latency, the sequencer uses a "fill before spill" policy to service MDC misses and the PP supports critical-word restart, meaning that it resumes operation immediately after the needed doubleword of data has returned from memory rather than waiting for the entire cache line to be delivered.

4.6.3 Outbox

The outbox performs outgoing message sends of behalf of the PP. It provides a high-level interface to the PI, NI, and IO units and allows the PP to send a message before the data associated with the message is ready, thus increasing the efficiency of the MAGIC macropipeline.

After constructing the outgoing MAGIC message header in its general-purpose registers, the PP executes a *send* instruction. The send instruction specifies two registers that contain the two doublewords of the outgoing message header. The contents of these registers pass to the outbox when the send is executed. In addition to these registers, several bits of the send opcode itself are also passed to the outbox. These bits are called the *message send type bits* and are used to specify the exact variation of send that the PP wishes to perform. Regardless of the specific send variation, the outbox determines the proper destination interface unit for the outgoing message, reformats the outgoing message header accordingly, and then passes the reformatted header to the destination interface unit for further processing.

In addition to handling the mechanics of outgoing sends, the outbox also provides flow control. If the destination interface unit is busy at the time the PP issues the send instruction, the outbox will accept the send, allowing the PP to continue processing, and then wait until the destination interface unit can accept the new outgoing message header. The outbox can process only one send at a time. If the outbox accepts a send but cannot immediately forward it to the destination interface unit, it will notify the PP that it is busy. Whenever the outbox is busy, the PP will stall if it attempts to issue another send. Once the outbox completes the original send, the outbox will accept the new send from the PP, allowing the PP to resume execution. This flow control between the PP and outbox is transparent to the PP programmer. The PP performs all instruction decoding and notifies the outbox when a send instruction has been issued. This decoupling isolates the outbox from the internal details of the PP implementation, particularly the effect of PP stalls caused by cache misses and the details of register forwarding.

4.7 Additional Issues

The primary goal of the MAGIC architecture is to provide efficient support for multiple protocols. The descriptions of the MAGIC functional units above illustrate how this architectural goal is implemented in a way that balances performance and flexibility. Three additional issues, however, warrant discussion: double-buffer operations, performance monitoring, and error handling.

First, MAGIC is designed to support both cache-coherent shared memory and message passing. As described above, the data transfer logic and control macropipeline support both communication paradigms using the same underlying hardware. The performance of message passing — in particular, the performance of bulk data transfer — can be increased significantly by the addition of some hardware features customized to this task. Section 4.7.1 explains the support for these so-called double-buffer operations and notes how they are used to improve block transfer performance.

Second, MAGIC includes efficient, flexible performance monitoring facilities. Performance monitoring support spans several MAGIC functional units and is presented in a unified form in Section 4.7.2.

Third, one goal of FLASH is to support large-scale shared memory systems in a realizable fashion. Although the operating system and other software needs to provide much of this support, a hardware foundation that provides comprehensive error detection and fault containment also is needed. Section 4.7.3 addresses this topic.

4.7.1 Accelerating Unaligned Block Transfers: Double-Buffer Operations

In normal operation, a functional unit handles data transfers by reading or writing only a single data buffer, beginning at doubleword zero in the buffer and continuing, for a cache line transfer, to doubleword 15. To make unaligned block transfers more efficient, however, the memory controller supports an additional type of data transfer operation — a *double-buffer operation* — that makes use of two data buffers. When a double-buffer operation is requested from the MC, the memory operation is performed using the two data buffers specified in the primary buffer number and wrap buffer number fields of the message header (see Figure 4.4). The MC supports only double-buffer read operations. That is, a double-buffer operation reads data from the node's memory and transfers the data into the two data buffers. The MC does not support a double-buffer write operation in which the data to be written to memory is retrieved from the two data buffers.

For single-buffer operations, bits 6:3 of the address supplied to the MC specify the starting doubleword offset that will be supplied to the memory system. The memory then returns the data beginning at this doubleword offset and the data is written into the single data buffer beginning at offset zero within the buffer and continuing sequentially through offset 15.

A double-buffer operation uses the address field differently. First, double-buffer operations make use of a *word-aligned* address rather than a doubleword-aligned address as is required for single-buffer operations. Second, in a double-buffer operation, bits 6:2 of the address supplied to the MC specify the starting *word* offset within the primary data buffer. These bits are masked to zero when supplying the address to the memory system. Thus, the memory system is always supplied with a cache line-aligned address. When the memory returns the data, the data is written into the primary buffer beginning at the word offset contained in bits 6:2 of the byte address and continuing sequentially through word offset 31 within the buffer. When word offset 31 of the data buffer is reached, subsequent data writes store the data into the wrap buffer, beginning at word offset zero and continuing sequentially thereafter.

Because this description of double-buffer operations may be confusing, Figure 4.13 shows an example double-buffer read from byte address 0x2FC8, which is doubleword-aligned. The contents of doublewords marked "X" within the data buffer are unaltered.

Double-buffer operations are not required for normal cache coherence processing. Rather, double-buffer operations are designed to provide efficient support for unaligned block transfers. In the absence of double-buffer operations, unaligned block transfers require the PP at the source or destination to align the incoming data manually using a series of read-modify-write (RMW) operations. With double-buffer support, the MC at the source effectively prealigns the message data before sending it to the destination, thereby dispensing with most of the read-modify-write operations.

Consider a block transfer from physical byte addresses 0x2FC8 through 0x31E7 at the source to byte addresses 0x0010 through 0x022F at the destination. The block is 0x220 (544_{10}) bytes long and spans five cache lines. Figure 4.14 depicts the block data layout in the source and destination memories (each rectangle in the figure represents all or part of a 128-byte cache line). To execute this block transfer, the source first allocates a series of data buffers and then performs the operations shown in Figure 4.15.

Example: Double-buffer memory read from byte address 0x2FC8

Memory Byte Address

X	X	X	X	X	X	X	X	X	2F80	2F88	2F90	2F98	2FA0	2FA8	2FB0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Primary Data Buffer Doubleword Offset

Memory Byte Address

2FB8	2FC0	2FC8	2FD0	2FD8	2FE0	2FE8	2FF0	2FF8	X	X	X	X	X	X	X
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wrap Data Buffer Doubleword Offset

Figure 4.13 Example double-buffer operation

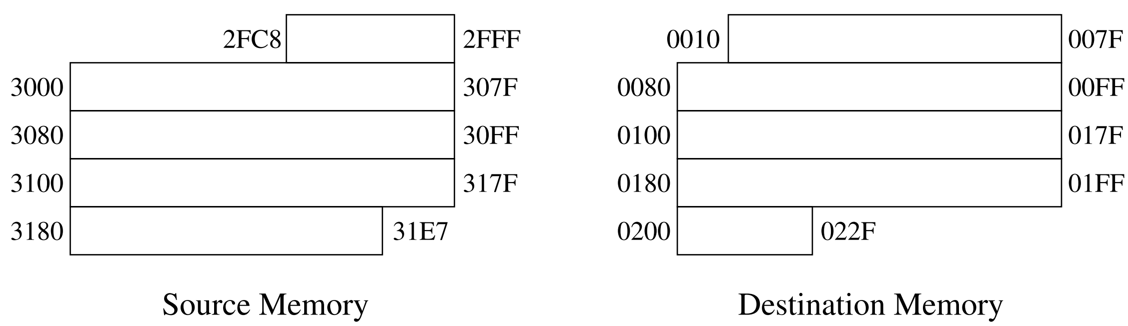


Figure 4.14 Memory layout of example unaligned block transfer

Figure 4.15 shows the source's double-buffer operations along with the primary buffer number and the addresses contained in the primary and wrap data buffers after each double-buffer operation completes. The figure also lists the message sends and the action the destination PP performs when it receives each message. This example assumes that the source PP has allocated five data buffers (B0 through B4) for use in the message transfer.

Although read-modify-write operations still are required at the destination for the first and last messages (buffers B0 and B4), the double-buffer operations eliminate the need for

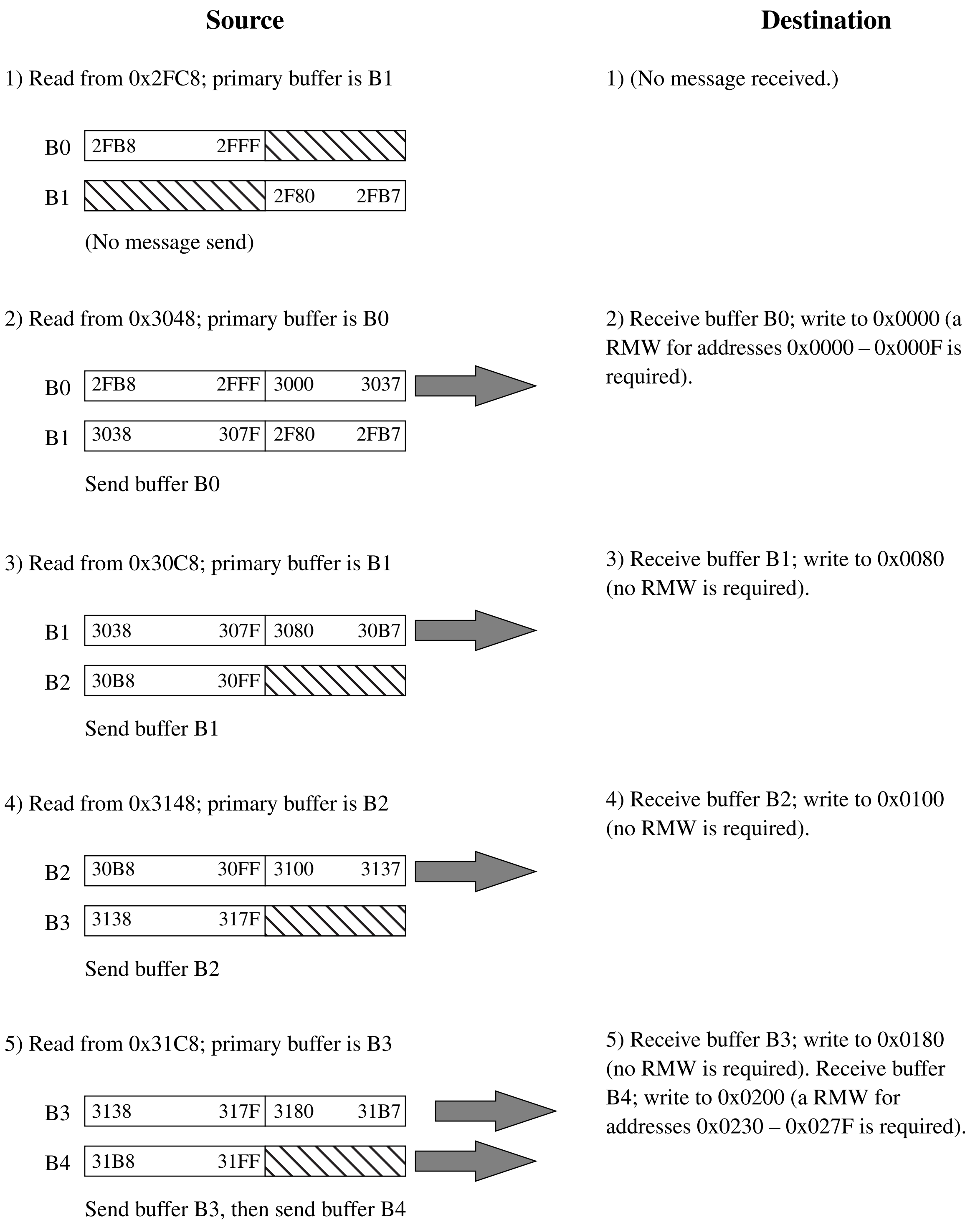


Figure 4.15 Unaligned block transfer double-buffer operations and message sends

any other read-modify-writes. For long block transfers, the additional latency of the first and last RMW operations becomes less important and the ability of the double-buffer operations to prealign the data at the source side yields a significant performance advantage. Of course, achieving the overall goal of data prealignment requires the source to know the destination target addresses before the block transfer begins. Whether the extra complexity and latency needed to communicate the destination alignment information to the source yields better performance depends on the total block length, PP processing overhead, and other factors. Further discussion of these tradeoffs is found in [41], which describes in detail the implementation of block transfer in FLASH.

4.7.2 Performance Monitoring

MAGIC supports performance data collection in two ways: hardware counters and sample-based handler invocation in the inbox. By using a combination of the two mechanisms, code executing on the protocol processor can perform performance analysis at varying levels of detail, processing overhead, and performance impact.

4.7.2.1 Hardware Counters

Almost every functional unit includes several hardware counters that can be read and written via the misc bus. These counters track activity of interest in the particular functional unit. The NI, for example, contains counters to track how many messages were sent and received on each of the four lanes. The DBA tracks how many buffers were allocated, how many reserved buffers were allocated, how many allocation attempts failed, and so on. Even the PP itself has hardware counters to count cache misses and similar information. The idea behind these counters is to provide a means to collect data that otherwise would be impossible to collect, or able to be collected only at a high performance overhead, using just the protocol processor itself.

Although the functional unit hardware counters can be read and written from the PP via the misc bus, the counters operate autonomously, collecting performance data as a normal part of each functional unit's operation. As such, the hardware counters have no impact on the performance or operation of the functional units in which they reside or on the protocol processor. To use the data in these counters, the PP must read the appropriate counter values using misc bus operations (load state instructions). Of course, reading the counters requires PP processing time and will have an impact on its performance, but the PP need not read the counters unless performance data collection is desired.

4.7.2.2 Sample-Based Handler Invocation

Just as the hardware counters allow collection of performance data that would be difficult or impossible to collect using the PP, certain other performance data can be collected *only* by the PP. This data tends to be protocol-specific, with the events of interest identifiable only in the course of executing a protocol handler. An example in a cache coherence protocol is the directory state of the cache line being requested. Only the protocol code has knowledge of the directory format, state encodings, and other information necessary to allow statistics to be collected on the number of times the line was clean, shared, dirty remote, etc.

The basic problem with this type of data is that the protocol processor itself must collect it. Unfortunately, using the PP to perform performance data collection necessarily incurs protocol processing overhead since the PP must execute performance data collection code in addition to the normal protocol handler. To maximize the flexibility and utility of PP performance data collection, then, both the frequency and the level of detail — and hence the overhead — of performance data collection on the PP must be controlled. As a result, MAGIC requires both a mechanism to decide *when* to collect performance data as well as a means to actually collect the data.

The MAGIC design adopts a sampling-based approach to decide when to collect performance data. In this approach, MAGIC collects performance data only once out of every N (user-specifiable) invocations of each handler. The premise of this technique is that over a long execution time, the behavior of a particular handler can be characterized accurately by collecting performance data during only a fraction of its invocations. This assumption is essentially the same as that underlying PC-based periodic sampling for application profiling, except that the sampling is applied on a per-handler basis rather than uniformly across all handlers. By varying N, the number of handler invocations between sampling, a tradeoff can be made between sampling frequency and the corresponding performance data collection overhead.

One implementation option for the sampling-based approach is to have the PP perform both the sampling decision and the data collection. Every handler that might want to collect performance data would be augmented with the necessary tests to decide whether to collect the data on the current invocation and, if the tests succeeded, the necessary data collection code would be executed along with the regular handler code. The advantages of this approach are that it requires no additional hardware and it allows for arbitrary sampling policies. The major drawback is that having the PP perform the sampling test on every handler is a significant overhead in itself. Many handlers are short, making the additional overhead of the sampling test unacceptable. For this reason, the MAGIC design off-loads the sampling decision logic into the jump table in the inbox, thereby relieving the PP of this task. Actual performance data collection is still performed in the PP since, as noted above, this information tends to be protocol-specific.

4.7.2.3 Jump Table Support for Sampling

Each jump table entry, as noted in Section 4.6.1.2, nominally contains a speculative memory operation indication, a mask and match field, and a starting PC value. To support sampling-based handler invocation, each entry also includes an additional starting PC value as well as a counter. Table 4.11 lists the complete contents of a jump table entry.

Table 4.11 Jump table entry contents

Field	Width	Description
Speculative flag	1	If set, a speculative memory operation will be initiated
Speculative R/W	1	Indicates whether the speculative memory operation is a read or a write
Mask	11	Mask field
Match	11	Match field
Sampling counter	16	Counter for sample-based handler invocation
Non-sampled PC	27	Location of start of normal handler
Sampled PC	27	Location of start of performance data collection handler

Whenever the inbox finds a matching jump table entry, it inspects the current value of the counter in the matching entry. If the value of the counter is non-zero, the non-sampled PC is supplied to the PP, the counter value is decremented by one, and the updated counter value is stored in the jump table. If, however, the counter value is zero, then the inbox will supply the sampled PC value to the PP. Because each jump table entry typically maps to a unique handler, and because the PP code itself controls how the counter values in the jump table are initialized, the net effect is to provide the per-handler, sampling-based performance data collection mechanism outlined above. The only change required to the protocol code is to write two versions of each handler that will participate in the performance data collection process: one version for the normal, non-sampling case and another version to perform performance data collection in the sampling case.

On the PP, the handler at the non-sampled PC (the non-sampled handler) simply contains the normal protocol code for the corresponding incoming message. The handler at the sampled PC (the sampled handler), however, must provide the code for both performance data collection and the normal protocol processing code. To avoid duplicating code, the sampled handler normally contains just performance data collection code and then, as its last action, jumps to the non-sampled handler. In addition to collecting performance data, the sampled handler also must reset the corresponding jump table entry counter to the desired initial value. In this way, not only is sampling performed on a per-handler basis but, by initializing the jump table entry counters appropriately, each handler may have a different sampling frequency. If no performance data collection is desired, the sampled PC and non-sampled PC fields in the jump table entry are both initialized to point to the non-sampled handler.

4.7.3 Error Handling

To provide robust operation, FLASH includes data integrity and fault containment features. Data integrity relies on error correcting code (ECC) and cyclic redundancy code (CRC) checksum techniques to verify that data remains valid as it is passed between major components. This hardware support is included to protect the integrity of communication between each of the primary system components: the processor, caches, main memory, and network. Once MAGIC has detected an error in any of these components, it propagates the error condition along with the corresponding data as the data is transferred between memory, the network, and MAGIC's internal data buffers. A processor therefore will discover an error when it eventually attempts to access the data across its bus.

While much of the responsibility for fault containment rests with the operating system, MAGIC must provide hooks that allow certain permission checks to be performed efficiently. The basic unit of fault containment in FLASH is called a *cell*, which is a relatively small cluster of processing nodes (typically 32 or fewer). The basic philosophy is that any fault occurring strictly within a cell should be contained within that cell. To contain faults, the fault containment mechanisms must consider all cases in which a cell is affected by a remote cell, including messages both received from and sent to the remote cell. To handle messages received from remote cells, the jump table includes support in hardware that allows different message handler code to be executed on the PP depending on whether the received message originated within the cell or from a remote cell (namely, the "cell" bit in the jump table comparison field; see Section 4.6.1.2). To handle messages sent to remote cells, MAGIC's processor and I/O interfaces include support to allow the PP to easily implement timeouts on all outstanding requests.

Although MAGIC contains several hardware features to aid in error handling and fault containment, the major challenge is not detecting faults, but recovering from them without crashing the entire machine. Indeed, the susceptibility of an entire cache-coherent shared memory system to crash when a minor error occurs traditionally has been a significant impediment to the construction of large-scale, distributed cache-coherent systems. The hardware features in FLASH provide a foundation on which a comprehensive fault detection, containment, and recovery mechanism can be built. A detailed discussion of the use of these features is beyond the scope of this dissertation. Instead, consult Rosenblum [87], which describes fault containment and recovery in distributed shared memory machines, with a focus on how this support is provided in FLASH.

4.8 MAGIC Implementation

From the outset, the goal of the FLASH design was to construct a prototype system. With the exception of the MAGIC chip itself, the other components on the FLASH node as well as the interconnection network were leveraged from others. Consequently, the primary focus of the prototype implementation effort has been the design and implementation of the MAGIC chip. This section discusses the implementation of MAGIC as well as some of the challenges encountered.

4.8.1 Implementation Technology

A major goal of the FLASH project, as described in Section 3.1.1, is to achieve performance competitive with a hardwired system. This goal demands a high-performance, single-chip implementation of the MAGIC design; overall system performance would suffer greatly otherwise. In addition, the complexity of the MAGIC design, as illustrated in the descriptions of the functional units in the sections above, adds to the implementation difficulty. As a result, deciding how to implement MAGIC was a challenge in itself.

Initially the MAGIC chip was to be implemented by the Supercomputer Systems Division of the Intel Corporation using a combination full-custom and standard cell methodology. Most of the control logic would have been synthesized and mapped into Intel's standard cell library. Datapaths, critical control structures, and certain memories, such as the jump table and data buffers, would have been implemented using a combination of standard cell and full-custom design techniques with hand layout. Intel engineers would have been responsible for almost all of the physical design effort, relieving that burden from the small design team at Stanford.

Unfortunately, for reasons that were largely managerial in nature rather than technical, collaboration between Intel and Stanford ended just as the MAGIC physical design effort at Intel was starting. An alternate means to fabricate the MAGIC chip was needed, and an ASIC implementation was desirable since it still would permit a single-chip, high-performance implementation.

LSI Logic's LCB500K standard cell process was selected [70]. This is a 3.3V process with 0.5µ drawn feature size (0.4µ L_{eff}) and three metal layers. At nominal operating conditions, a two-input NAND gate with a fanout of four has a propagation delay of approximately 170ps. Given these characteristics, the same clock rate as the Intel-fabricated design targeted — 100 MHz — seemed achievable in the ASIC process as well.

4.8.2 Implementation Flow

Design work began with the development of a Verilog RTL model of MAGIC. Each functional unit was partitioned into datapath and control logic. The control logic sections were synthesized using Synopsys and mapped to the LSI standard cell library. In contrast, datapaths were largely hand-instantiated. That is, cells from the LSI library were instantiated explicitly in the datapath RTL and very little or no synthesis was performed in mapping the datapath RTL into LSI primitives. Initial timing analysis from Synopsys was used at this stage to improve the timing of critical paths in both the datapaths and control logic, usually by recoding the RTL or, in some cases, by hand-instantiating gates in the control logic. Overall, approximately 10% of the control logic consists of hand-instantiated gates.

With the exception of the data buffers, all memories were generated using LSI's memory compiler. None of the existing LSI memories provided the needed multiporting for implementing the data buffer memory array. As a result, the data buffer array was designed at Stanford as a full-custom cell and later integrated into the chip layout at LSI.

Once all synthesis was complete, scan chains were inserted and the entire gate-level netlist of the MAGIC chip was transferred into the LSI CMDE system, LSI's proprietary ASIC physical design environment. Several major tasks were performed in CMDE. First was development of a full-chip floorplan that guided placement and aspect ratios for the various functional units. Next was hand-layout of several timing- or area-critical datapaths and control logic blocks. LSI's datapath place and layout (DPPL) tool and routers were used for this purpose. The remaining cells in the design were placed using the normal standard cell placement tools and then merged with the DPPL-generated blocks.

At this point, the layout was ready for routing. The regular suite of LSI routers succeeded in routing all but approximately 200 of the more than 85000 nets in the design. Manual routing of the remaining open nets completed the process. The fully-routed design then was sent to LSI for fabrication and also used to extract back-annotated delay information for post-routing timing analysis. LSI merged the memories into the layout, performed layout-versus-schematic checks, and completed other backend processing tasks.

The resulting MAGIC die is 16mm x 16mm and contains 250K actual gates in addition to 200Kbits of on-chip memory, or about 750K two-input gate equivalents in total. Surrounding the core is the pad ring, containing 700 pads of which 451 are signals and 249 are power and ground. An HTTBGA cavity-down ball grid array, manufactured by IBM, is used to package the die.

4.8.3 Implementation Challenges

The major challenges in completing the MAGIC logical and physical design were timing and design verification. Achieving the cycle time target is challenging in most high-speed designs and MAGIC was no exception. Although the 100 MHz goal is likely achievable, the initial design operates under worst-case temperature, voltage, and process conditions at 77 MHz. Further efforts to reduce the cycle time were possible, but would have caused additional delays in sending the chip to LSI for fabrication and hence were postponed.

Most of the functional units did meet the 10ns cycle time goal. The major exception was the protocol processor. Three critical paths exist in the PP and these govern the overall chip cycle time. First is the MAGIC instruction cache access path. Physically long connections to the four cache RAMs coupled with a raw RAM access time of 7ns are the problem here. Second is the MAGIC data cache access path, which suffers from delays in driving the cache address off-chip and then waiting for the data to propagate back on-chip. Third is the global PP stall signal used to pause PP execution when a cache miss or other exceptional condition occurs. The stall signal is generated by and must be distributed to multiple locations in the PP logic.

Design verification, like timing, was also a challenge, particularly given the complexity of the MAGIC design both in terms of raw gate count and functionality. In fact, verification activities occupied more time than the development of the RTL and were complicated by the need to verify not only the MAGIC hardware but also the protocol code running on the PP. Directed diagnostics, random stress tests, and actual applications using real protocol code all were used in this effort. Error detection relied on self-checking diagnostics and “snoopers” in the Verilog code that detected incorrect operation of the RTL. This approach is similar to the one used for the DEC Alpha 21164 microprocessor, as described in [58].

4.9 Summary

The MAGIC architecture balances flexibility and performance. The key architectural feature in support of this goal is the separation of data transfer from control (protocol) processing. By separating these two tasks and providing dedicated, specialized hardware for each, the overall performance of the chip is maximized without losing the flexibility to support multiple protocols and a variety of performance monitoring capabilities.

Separation of data and control information begins at the interface units — the PI, NI, and IO — which handle the connection of the MAGIC chip to the R10000 compute processor, interconnection network, and local PCI bus. Incoming messages are split into message data and message headers. The message data is passed to the data transfer logic and the message headers, containing control information, are passed to the control macropipeline. For outgoing messages, this process is reversed, with the interface units taking outgoing headers and combining them with any associated outgoing data to form complete outgoing messages suitable for transfer off-chip.

The control macropipeline, composed of the inbox, protocol processor, and outbox, forms a closely-coupled mechanism that uses a combination of software and hardware to provide efficient protocol processing. The inbox selects incoming message headers and passes them through the jump table, which determines the starting PC value at which the protocol processor begins processing the message header and optionally initiates a speculative memory operation for the address contained in the message header. From the inbox, the message headers enter the protocol processor, the heart of the MAGIC design. The PP is a dual-issue, statically-scheduled RISC core, with instruction set extensions to accelerate common protocol operations and allow the PP to communicate with the other functional units. Outgoing message headers are passed to the outbox, the final stage of the control macropipeline, which manages header format conversions and flow control between the PP and the destination interface units.

Data transfer is largely a protocol-independent task, and therefore is implemented in a completely hardwired fashion in the data transfer logic. Hardwired data transfer logic allows for high-bandwidth, low-latency data movement and permits extensive data pipelining. The data buffers are the central component of the data transfer logic. Each data buffer is one cache line in size and is used to stage data as it flows from source interface to destination interface. Per-word valid bits and an error bit provide additional support for data pipelining and error detection and propagation to ensure that corrupted data is not used. Because the data buffers are used by several MAGIC functional units, the data buffer allocator is needed to manage data buffer allocation and usage tracking. Completing the data transfer logic is the memory controller, which controls the node's local memory system, providing memory access on behalf of the protocol running on the protocol processor as well as servicing MAGIC instruction and data cache misses.

LSI Logic's LCB500K 0.5µ standard cell ASIC process was used to implement the MAGIC design. The chip is 16mm on a side, contains approximately 250K actual gates and 200Kbits of on-chip SRAM, and has 451 signal pads and 700 total pads.

Chapter 5

MAGIC Operation: Protocols

The goal of MAGIC is to perform efficient protocol processing. This chapter discusses three protocols developed for MAGIC, demonstrating how the architectural features and functional units described in Chapter 4 are actually used. First, this chapter presents the data structures and an outline of the protocol processor code implementation of two cache coherence protocols: the bitvector/coarsevector protocol and the dynamic pointer allocation protocol.

Second, this chapter describes the interaction between MAGIC and the operating system. Use of the node controller to perform relatively complex low-level operating system services usually is impossible because of the hardwired implementation of the node controller in typical multiprocessor systems. In MAGIC, however, the presence of the protocol processor offers tremendous opportunity for performing services on MAGIC that are traditionally performed in the operating system on the compute processor. Section 5.5 describes the basic services provided and the mechanism for invoking them.

The cache coherence protocols and MAGIC–OS interaction mechanisms described in this chapter are just some of the possible protocols that can be provided on the system. These protocols also are the ones used for the system performance analysis presented in Chapter 6, and details of the protocol implementations are useful in interpreting the performance results. Other protocols certainly will be developed for the machine; for example, [45] discusses the development of a cache-only memory architecture (COMA) protocol [55] and an implementation of the scalable coherent interface (SCI) protocol [10, 39, 91] for FLASH.

5.1 Basic Cache Coherence Protocol Operations

A directory-based cache coherence protocol defines both a directory format and a set of messages that are exchanged among the processing nodes. This section presents some basic protocol transactions and illustrates the messages that are exchanged as a memory transaction proceeds. Although the basic operations are common to most cache coherence protocols, the messages exchanged tend to be protocol-specific. The message types discussed next are those exchanged by the bitvector/coarsevector and dynamic pointer allocation protocols, and are a component common to both protocols. The two protocols differ in the directory entry formats they maintain. Section 5.2 and Section 5.3 discuss the bitvector/coarsevector and dynamic pointer allocation directory entry formats in detail.

5.1.1 Directory Usage

To achieve tradeoffs among scalability, implementation complexity, and directory memory overhead, different protocols employ varying directory entry formats. Fundamentally, however, regardless of its particular format, a directory entry tracks two pieces of information: the state of the associated cache line and a list of sharers.

Abstractly, a cache line can be in one of three basic states:

- *Clean* — the line is not held in any cache in the system. The only copy of the line in the system is the one stored in the node's local memory.
- *Shared* — the line is held in a shared (read-only) state in one or more caches in the system. The sharer list indicates the set of caches containing the line. The copy of the line in the node's local memory is identical to the copies of the line held in the sharers' caches.
- *Dirty* — the line is held in an exclusive (read-write) state in exactly one cache in the system. The sharer list, which in the dirty case contains just one entry, indicates the cache containing the line. The most up-to-date copy of the line is held only in the dirty processor's cache; the copy in the node's local memory is stale.

When an incoming message arrives, the current state of the line along with the sharer list are used to determine how to update the directory entry and what outgoing messages need to be sent. The following subsections describe this process in the context of servicing cache read misses and cache write misses.

5.1.2 Servicing a Cache Read Miss: The Get Operation

When a compute processor executes a load instruction and does not find the load address in its cache, it issues a read miss on the SysAD bus. The MAGIC processor interface receives the read miss and translates it into what is called a Get request, meaning a request to retrieve a cache line in a shared (read-only) state. In its simplest form, the Get request targets a line in the Clean or Shared directory state and is satisfied by the sequence of messages illustrated in Figure 5.1:[1]

1. The local MAGIC chip on the requesting node (labelled "R" in the figure) forwards the Get to the corresponding home node, that is, to the node containing the physical memory for the address.

---

1. An even simpler case occurs when the home node is the same as the requesting node. In this situation the Get is serviced from the requesting node's local memory and no messages are exchanged.

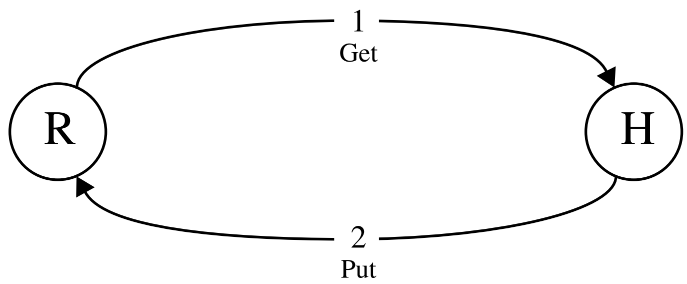


Figure 5.1 Messages exchanged in servicing a simple Get operation

2. The home node, labelled "H" in the figure, receives the Get and consults the associated directory entry, which indicates that the state of the line is Clean or Shared. The directory state is updated to Shared and the requesting node is added to the sharing list. A Put message containing the data is sent to the requesting node.
3. The Put is received at the requesting node and echoed on the SysAD bus, completing the transaction. The data is placed into the R10000 cache in a shared (read-only) state.

A more complicated case occurs when the Get request arrives at the home node and the directory indicates that the cache line is in the Dirty state. This case is depicted in Figure 5.2 and requires several messages to be exchanged:

1. As in the previous case, the Get is forwarded from the requesting node to the home node.
2. The home node finds the line in the Dirty state and forwards the Get to the Dirty node, that is, to the node holding the line exclusively in its cache.
3. The dirty node, labelled "D" in the figure, receives the Get and responds by retrieving the current copy of the cache line from the local processor's cache and changing the state of the line in the processor cache from exclusive (read-write) to shared (read-only). In addition, the dirty node sends a Put message with the data to the requesting node and a Sharing Writeback (SWB) message, also containing the data, to the home node.
4. The requesting node receives the Put and echoes it on the SysAD bus, completing the transaction. The data is placed into the R10000 cache in a shared (read-only) state.
5. The home node receives the Sharing Writeback, writes the data to its local memory, and updates the directory state to Shared with both the original dirty node and the requesting node on the sharing list.

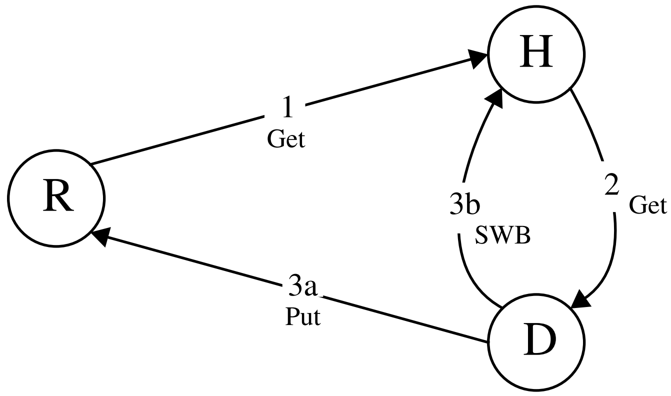


Figure 5.2 Messages exchanged in servicing a three-hop Get operation

This type of Get operation is called a *three-hop case* in reference to the number of network traversals required before the Put arrives at the requesting node and satisfies the original Get.

5.1.3 Servicing a Cache Write Miss: The GetX Operation

Just as a cache read miss results from an R10000 executing a load instruction and not finding the address in its cache, a cache write miss results when a store instruction is executed rather than a load. The MAGIC processor interface translates the cache write miss from the SysAD bus into a GetX request, meaning a request to retrieve a cache line in an exclusive (read-write) state. Again like the Get request flow described in Figure 5.1, the simplest form of GetX request is one in which the GetX targets a line in the Clean state. The messages exchanged in this operation are similar as well, and are depicted in Figure 5.3:

1. The local MAGIC chip on the requesting node forwards the GetX to the corresponding home node.
2. The home node receives the GetX and consults the associated directory entry, which indicates that the state of the line is Clean. The directory state is updated to Dirty and the requesting node is added to the sharing list. A PutX message containing the data is sent to the requesting node.
3. The PutX is received at the requesting node and echoed on the SysAD bus, completing the transaction. The data is placed into the R10000 cache in an exclusive (read-write) state.

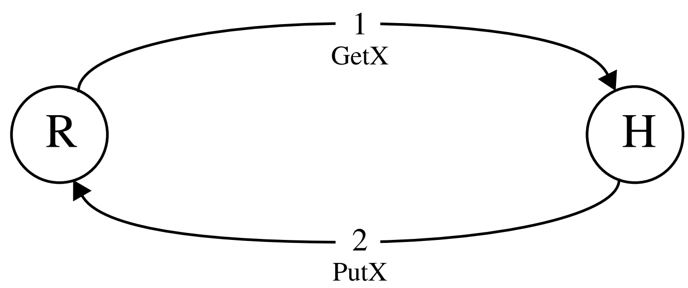


Figure 5.3 Messages exchanged in servicing a simple GetX operation

A more complicated situation occurs when the target of the GetX is in the Shared state, indicating that the line is held read-only in several caches in the system. Before the GetX can be completed, the cache line must be invalidated, or removed, from all caches on the sharing list. Figure 5.4 illustrates this transaction:

1. The local MAGIC chip on the requesting node forwards the GetX to the corresponding home node.
2. The home node receives the GetX and consults the associated directory entry, which indicates that the state of the line is Shared. The home sends invalidation requests to all nodes on the sharing list (labelled "S" in the figure), updates the state of the line to Dirty, and revises the sharing list to contain only the requesting node. A PutX message with the data is sent to the requesting node.[2]
3. The requesting node receives the PutX, echoes it to the SysAD bus, and places the data in the processor cache in an exclusive state. The data may be used immediately even though all the invalidate acknowledgments have not been received.
4. The sharers receive the invalidate requests. Each node invalidates the line from its local processor's cache and sends an invalidate acknowledgment to the home node.
5. When all invalidate acknowledgments have been received at the home, an AcksDone message is sent to the requester, completing the transaction.

2. The case described makes use of a relaxed consistency model because the PutX is sent to the requester before all invalidation acknowledgments have been received at the home. A delayed consistency model also is supported in which the PutX is deferred until all invalidation acknowledgments have been received.

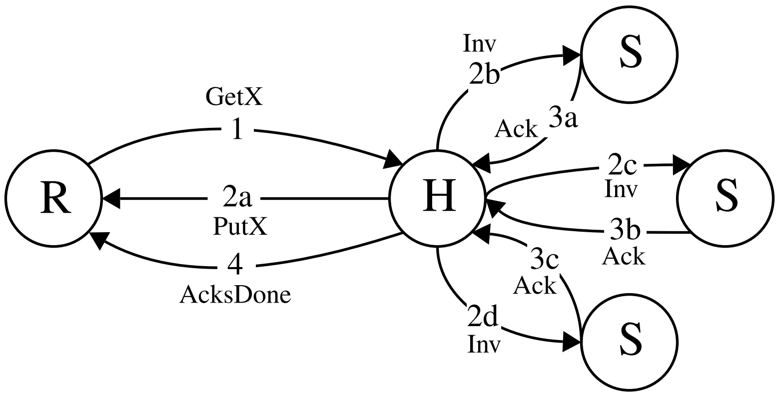


Figure 5.4 Messages exchanged in servicing a GetX operation to a Shared line

5.1.4 Protocol Complications

A cache coherence protocol must handle numerous other operations in addition to the ones described above. Other directory states can occur when a Get or GetX request reaches the home node, and entirely new types of operations also must be supported. These include I/O operations, uncached operations, and support for synchronization primitives, such as the “load linked” and “store conditional” instructions provided in the MIPS instruction set. Although these operations are important components of a protocol and, particularly in the case of synchronization support, can have a measurable impact on system performance, further discussion is beyond the scope of the protocol overview presented in this chapter.

One issue that warrants further discussion, however, is protocol races. Even within the Get and GetX operations, complications often occur that were ignored in the previous discussion. Among the most common complications are protocol races in which one request is still in progress when another request for the same cache line arrives. In the GetX to a shared line operation depicted in Figure 5.4, for example, another request for the same cache line may arrive at the home node after the PutX is sent to the requesting node but before all invalidate acknowledgments have been received. The second request could even be from one of the original sharers that has invalidated the line from its cache and is once again requesting it. Handling such protocol races, often called transient conditions, greatly complicates the implementation of any cache coherence protocol and provides another reason to opt for a flexible protocol processor, since the flexibility allows protocol bugs to be corrected even after the node controller has been fabricated.

5.2 Bitvector/Coarsevector Protocol

The characteristics of a protocol — its scalability, processing overhead, and implementation complexity — are a function of both the directory format it uses and the message types it requires. This section and Section 5.3 delve into some of the details of the bitvector/coarsevector and dynamic pointer allocation protocols: the specific directory formats and examples of the actual MAGIC protocol processor code sequences used to process the basic cache coherence operations presented earlier.

On a spectrum of implementation complexity and processing overhead, the bitvector/coarsevector protocol lies at the simple, low overhead end. It employs a simple directory format, exchanges relatively few message types, and requires comparatively small amounts of processing in response to incoming messages. As a result, the protocol is straightforward to implement and has low protocol processing overhead. The ease of implementation also makes the bitvector protocol an attractive choice for use in a hard-wired node controller. DASH, for instance, uses the bitvector protocol, although a simpler version than the one presented here, and the SGI Origin 2000 system uses a variation of the bitvector/coarsevector protocol.

The price of bitvector's implementation ease is its lack of scalability. As will be discussed below, the memory overhead of the bitvector protocol grows with the square of the number of processors in the system. Consequently, the protocol is suitable only for small-scale machines of perhaps a few tens of processors. By sacrificing some of its simplicity, however, the bitvector protocol can be extended for use in larger systems. In this mode, the protocol is called coarsevector, in reference to its imprecise or "coarse" tracking of sharers. Thus, bitvector/coarsevector is really a two-stage protocol: the first stage, in which the bitvector mode is used, is suitable only for small processor counts; as the system size grows the second stage, coarsevector mode, must be substituted.

5.2.1 Bitvector/Coarsevector Directory Format

A directory entry must track two types of information: the state of the associated memory line and the sharers. As discussed in Section 2.1.3, the basic idea of the bitvector protocol is to maintain an array of presence bits for each cache line, with one presence bit for each processor in the system. State information is kept in a separate bitfield. In practice, the actual directory entry format is a little more complicated; Figure 5.5 depicts the format of the bitvector directory entry used in the implementation of this protocol for FLASH. The figure shows a single node's local memory (viewed as an array of 128-byte cache

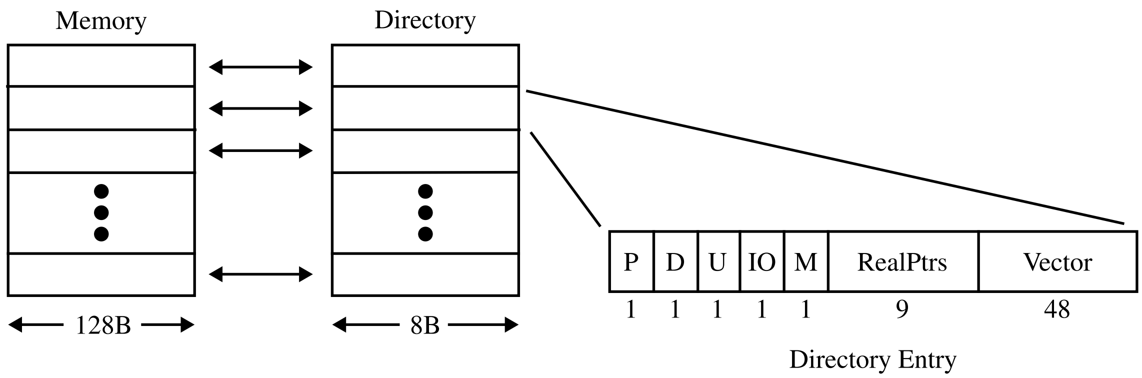


Figure 5.5 Bitvector/coarsevector memory and directory organization

lines), the corresponding array of 8-byte directory entries, and, on the far right, the format of an individual directory entry.

The various bits in the directory entry are used as follows:

- The *Pending* bit (P) is set if the cache line is in a transient state. Requests for a cache line whose directory entry's pending bit is set typically are negatively acknowledged until the transient condition is resolved and the pending bit is cleared.
- The *Dirty* bit (D) is set if the line is held exclusively in a single processor's cache; if clear, the line is shared among zero or more processors.
- The *Upgrade* bit (U) is used in maintaining the proper memory consistency model for the cache line. The bitvector protocol supports both processor consistency and release consistency. Consistency models are discussed fully in Gharachorloo [35].
- The *I/O* bit (IO) is set if the line is held in the one-line cache in a node's I/O subsystem (see Section 4.5.3) rather than in a node's compute processor's cache.
- The *MoreInvals* bit (M) is set if an exclusive request (GetX) has been made for the line and additional invalidation requests still need to be sent.
- The *RealPtrs* field contains a count of the number of sharers (that is, a count of how many bits are set in the Vector field).
- The *Vector* field is the actual presence bit vector. It tracks which nodes are sharing the cache line or which node holds the line exclusive.

In general terms, the first five fields above comprise the directory state and the vector field is the presence bit vector. The RealPtrs field is just an optimization; a count of the number of bits set in the vector field is required for certain protocol operations but computing the count directly from the contents of the vector field is inefficient to perform on the protocol processor.

Use of the bitvector directory entry is straightforward. When the PP handler begins processing an incoming message, it first retrieves the associated directory entry. Because each cache line in the local memory has an associated directory entry, the address of the proper directory entry can be generated by scaling the incoming address by the size of the directory entry (eight bytes) and then adding the result to the base address of the directory entry array. Once the directory entry has been loaded, the handler code inspects various state bits to determine how to update the directory entry and what outgoing messages need to be sent.

5.2.2 Bitvector Scalability Limitations

The primary advantage of the bitvector protocol is its simplicity. All state information and sharing information is contained in a single 64-bit directory entry. State checks require only a few bit tests, and modifying the vector field to add or delete sharers can be performed quickly using simple shift and logical instructions. As a result, the bitvector protocol implementation is characterized by short protocol handlers, helping to minimize protocol processing overhead. For small-scale systems, then, bitvector/coarsevector is an excellent protocol choice. Unfortunately, the advantages of the bitvector/coarsevector protocol are attainable, in general, *only* for small-scale systems. In particular, the protocol suffers from a severe scalability limitation — the memory required by the directory — as the system size grows.

Because a unique vector bit is needed for each processor, the size of the vector field must grow linearly as the number of processors is increased. In addition, the protocol requires one directory entry for each memory line in the system. Consequently, as more processors are added, the number of directory entries also grows linearly. Overall, therefore, the aggregate directory storage space for all processors in the system — the total number of directory entries multiplied by the size of each entry — grows proportionally to the *square* of the processor count because both the number of directory entries and the size of each entry grow linearly as more CPUs are added.

Thus, bitvector mode is not easily extended to larger processor counts. Extending the vector field as more processors are added is infeasible beyond a few tens of processors because the size of the directory rapidly would approach the size of the memory it is being used to track, clearly an unacceptable situation. Even if the directory overhead increase were tolerable, the huge size of the vector field would negate much of bitvector mode's simplicity and low protocol processing overhead. Transition into coarsevector mode, presented in the next section, allows the size of each directory entry to remain fixed as the number of processors increases, but comes with a loss of precise sharing information and the associated protocol processing overhead increase and performance loss.

5.2.3 Coarsevector Mode

When the total system processor count exceeds the length of the vector field, the basic principle of allocating one bit in the vector field for each processor is no longer feasible. Somehow the use of the vector field must be extended to allow it to track sharing information for larger processor counts. The most common approach to solving this problem is to redefine each bit in the vector field to represent several processors rather than a single processor. This approach is said to make the bits in the vector field "coarse" (in that each represents multiple sharers) and gives rise to the "coarsevector" terminology — "bitvector" refers to the use of the protocol in its most efficient mode of one processor per vector bit and "coarsevector" refers to the mode in which multiple processors are mapped to a single vector bit. If each bit in the vector field represents C processors (C is the coarseness factor), then bit i of the vector field represents processors $(C \bullet i)$ through $(C \bullet (i + 1) - 1)$ and, in many cases, messages sent to one of these processors must be sent to all of them.

For the specific 48-bit size of the FLASH vector field, the protocol uses one bit per processor for processor counts up to 48. For processor counts over 48, each bit in the vector field represents a power of two number of processors, with the particular coarseness factor determined by the maximum system processor count. A system with 256 processors, for example, would need a coarseness factor of eight. (Ideally, each bit in the vector field would represent 256 / 48 = 5.3 processors, but to achieve efficient protocol code the coarseness factor needs to be a power of two, so 5.3 must be rounded up to 8, the next higher power of two.)

Although the coarsevector modification allows the basic bitvector protocol to be employed for large processor counts, the mapping of multiple processors to one vector bit means that the protocol can no longer retain precise sharing information. In other words, in the coarsevector mode, a single vector bit maps to many processors and there is no means to determine the exact subset of these processors actually caching the associated memory line. As a result, the protocol handlers must be conservative and assume that all processors mapped to a particular vector bit are caching the memory line. This assumption is necessary for correctness but leads to significantly longer protocol processing handlers for a given protocol operation and requires many more intra-node messages to be sent. In the GetX case described in Section 5.1.3, for instance, invalidation requests must be sent to all processors represented by the vector bits rather than to the specific processors actually caching the memory line. In a 256-node system, up to eight times more invalidation requests might be generated than are actually needed, leading to higher protocol processing overhead at both the home node and sharing nodes. The results to be presented in Chapter 6 demonstrate that the additional overheads of coarsevector can result in significant performance loss even at as few as 64 CPUs.

Hence, for small-scale systems — say, fewer than 64 CPUs — the bitvector scheme has an acceptable directory memory overhead and good performance. For systems of more than roughly 100 processors, however, the coarsevector mode must be used and overall system performance likely will degrade. As a result, the coarsevector mode is unlikely to achieve best performance on larger systems. Instead, larger systems benefit from a protocol with an inherently more scalable means for tracking sharing information, such as the dynamic pointer allocation protocol described next.

5.3 Dynamic Pointer Allocation Protocol

As indicated above, larger systems require a protocol whose directory memory overhead is comparable to coarsevector yet which can continue to maintain precise sharing information as the processor count increases. The dynamic pointer allocation protocol, proposed by Simoni [95], is designed to meet these goals.

5.3.1 Dynamic Pointer Allocation Directory Format

Like the bitvector/coarsevector protocol, the dynamic pointer allocation protocol maintains one directory entry for each cache line in a node's local memory. Unlike the bitvector protocol, however, the dynamic pointer allocation directory entry does not contain a presence bit vector or other sharing information. Instead, all sharing information is maintained in a linked list, and only a pointer to the start of the list is kept in the directory entry. Figure 5.6 depicts the two dynamic pointer allocation data structures: the *head link array* and the *pointer-link store*. The head link array is an array of directory entries, called head link entries, containing state information and a pointer to a linked list of sharers. The linked list of sharers is maintained in the pointer-link store, which is a pool of pointer-link entries shared by all head link entries. Pointer-link entries available for use (that is, free) are themselves maintained in a linked list, and the *free list pointer* points to the start of this list.

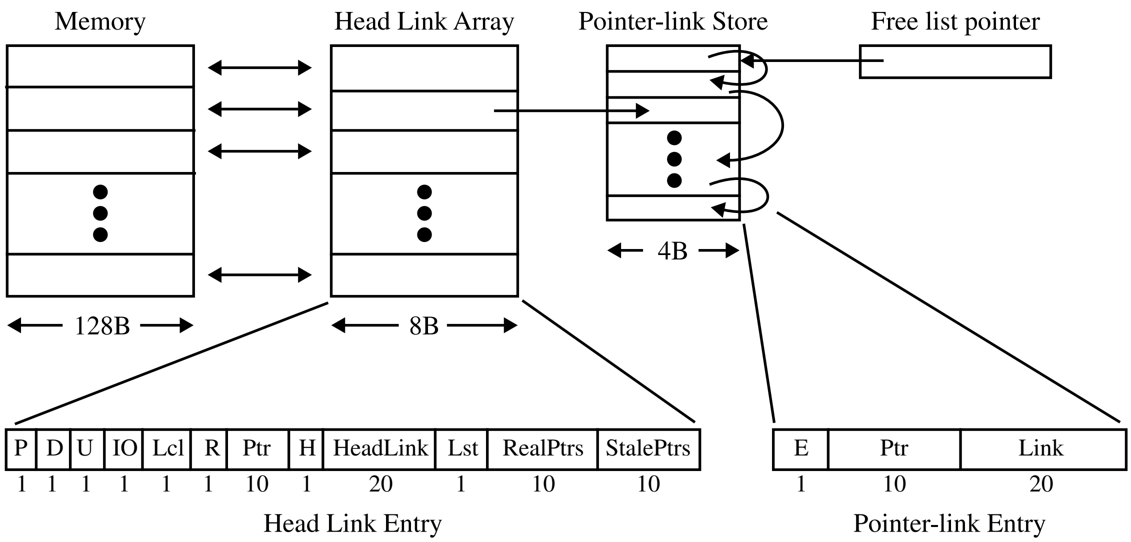


Figure 5.6 Dynamic pointer allocation memory and directory organization

The fields of the 64-bit head link entry are used as follows:

- The *Pending* bit (P), *Dirty* bit (D), *Upgrade* bit (U), and *I/O* bit (IO) are used in the same way as for the bitvector/coarsevector protocol; see Section 5.2.1 for details.
- The *Local* bit (Lcl) is set if the local compute processor is caching the memory line. Local caching is a common situation. This bit avoids the need to explicitly add the local processor to the linked list of sharers.
- The *Reclaim* bit (R) is set when the supply of free pointer-link entries is exhausted and pointer-link entries are being reclaimed. Section 5.3.2.2 discusses this problem in greater detail.
- The *Pointer* field (Ptr) contains the number of the first sharing node or of the dirty node if the Dirty bit is set. Extra space remained in the head link entry after all state information was assigned, and storing the first sharer in the head link entry improves performance by avoiding the need to allocate a pointer-link entry for the first sharer.
- The *HeadPtr* bit (H) is set if the Pointer field is valid.
- The *HeadLink* field points to the first pointer-link entry of the sharing list.
- The *List* bit (Lst) indicates whether the head link entry has a list of sharers. If this bit is set, then the HeadLink field is valid and points to the first pointer-link entry of the sharing list.
- The *RealPtrs* field stores a count of the number of sharers on the sharing list.

- The *StalePtrs* field is used to optimize the searching of the sharing list when sharers are added or removed; further discussion of this optimization is found in Heinrich [45].

The 32-bit pointer-link entries contain just three fields, used as follows:

- The *End* bit (E) is set if the pointer-link entry is the last element of the linked list of sharers; essentially it indicates whether the Link field is valid.
- The *Pointer* field (Ptr) contains the number of the sharing node.
- The *Link* field points to the next pointer-link entry in the linked list of sharers.

Initially, all pointer-link entries are free and are linked into a single free list. All head link entries indicate that the associated memory line is in the Clean state and that no sharers are present. Thereafter, two basic operations need to be performed: adding a new sharer and transitioning a line from the Shared state to the Dirty state.

When a new sharer needs to be added, an available pointer-link entry is removed from the free list, its Pointer field is initialized to the number of the sharing node, and the associated head link entry's HeadLink field is set to point to the new pointer-link entry. The previous contents of the head link's HeadLink field are copied into the pointer-link entry's Link field to complete the process. New sharers are added to the front of any existing sharer list to avoid having to traverse the entire list before adding the new pointer-link entry. In addition, the last pointer-link entry in the sharing list has its End bit set and its Link field points back to the head link entry at the start of the list. Thus, the sharing list is actually a circular list in that the last pointer-link entry points to the head link entry. This characteristic is used in handling pointer-link store exhaustion, as discussed in Section 5.3.2.2. Also, as mentioned above in the description of the head link entry fields, one optimization is made in this process: the first sharer is stored in the head link entry itself, using the Pointer field, rather than in a pointer-link entry. In all cases, the RealPtrs field in the head link entry tracks the current sharer count, so it is incremented each time a new sharer is added.

To illustrate the layout of the sharing list, Figure 5.7 depicts a head link entry with an attached sharing list of four pointer-link entries. As shown, the initial sharer is stored in the head link entry itself (node 12 in the example), and the remaining four sharers (nodes 9, 17, 3, and 14) are stored in four pointer-link entries. The head link entry points to the first pointer-link entry, and the last pointer-link entry (node 14) has its End bit set and points back to the head link entry.

Transition of a line from the Shared to Dirty state occurs in the course of processing GetX messages and requires that all sharers be sent invalidation request messages. This

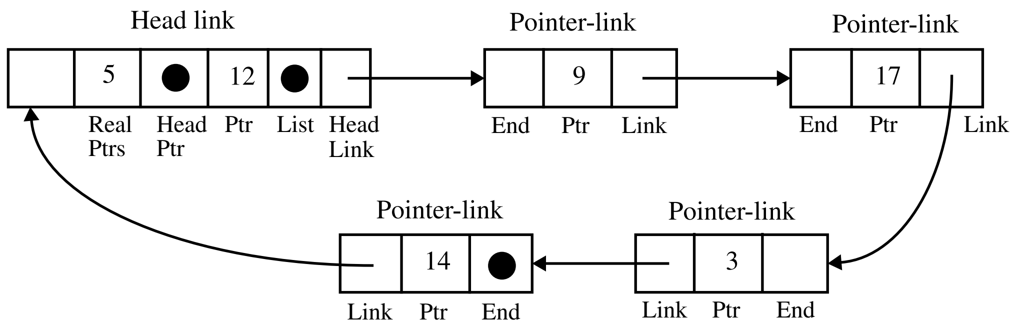


Figure 5.7 Example of head link entry with attached sharing list

process is accomplished by traversing the chain of pointer-link entries attached to the head link entry and sending an invalidation request to each node in the sharing list. At the same time as the invalidation requests are sent, the associated pointer-link entry is returned to the free list. Once all invalidation requests have been sent, the state transitions to Dirty and the Pending bit is set to indicate that the line is in a transient state waiting for invalidation acknowledgments. Upon receipt of each invalidation acknowledgment, the RealPtrs field is decremented. When RealPtrs reaches zero, all invalidation acknowledgments have been received and the Pending bit in the head link entry is cleared.

5.3.2 Dynamic Pointer Allocation Complications

Use of a linked list rather than a fixed-size presence bit vector allows sharing information to be kept precisely even for very large processor counts. An arbitrary number of sharers can be accommodated by allocating new pointer-link entries and extending the linked list of sharers attached to a head link entry. Having a common pool of pointer-link entries avoids the need to statically dedicate large numbers of pointer-link entries to each head link entry, and thereby limits the growth in directory memory overhead as more processors are added.

Thus, the dynamic pointer allocation protocol overcomes the two most serious drawbacks of the bitvector/coarsevector protocol: lack of precise sharing information in coarsevector mode and unacceptable directory memory size growth in bitvector mode. Unfortunately, dynamic pointer allocation has two drawbacks of its own: the need for replacement hints and the ability to cope with exhaustion of free entries in the pointer-link store. Although these drawbacks do not impact the overall scalability advantages versus bitvector/coarsevector, they do add implementation complexity and processing overhead.

5.3.2.1 Replacement Hints

When an R10000 compute processor requests a cache line in an exclusive (read-write) state, the R10000 can evict the line from its cache only after first issuing a writeback to the local MAGIC chip. For a line requested in the shared (read-only) state, however, the R10000 is free to evict the line from its cache without notifying either the local MAGIC chip or the home node. If a processor continually reads several different cache lines whose addresses are such that they all map to the same line in the R10000 data cache, then a cache conflict occurs. The result is that each of the conflicting cache lines is read into the cache in a shared state, evicted when one of the other cache lines is read, and then read into the cache again.

Within the memory system, each of the home nodes of the conflicting addresses receives a stream of repeated Get requests, all for the same address. These requests are serviced as normal: the state of the line is checked, found to be Shared, and the requesting processor is added to the sharer list. Because of the continual stream of Get requests, the same processor is added to the sharer list many times. In the bitvector/coarsevector protocol, this situation does not lead to difficulty because adding a processor to the sharing list just means to set the corresponding bit in the presence bit vector. If this bit is set already, setting it again is unnecessary, of course, but otherwise has no adverse effects.

In the dynamic pointer allocation protocol, however, adding a new sharer requires the home node to allocate a new pointer-link entry and attach it to the head link's linked list of sharers. The problem with the continual sequence of Get requests is that it causes the same processor to be added to the sharing list many times, so the length of the sharing list can grow without bound even though no additional processors are actually sharing the associated cache line. Worse, each time the requesting processor is re-added to the sharing list, a new pointer-link entry must be removed from the free list, thereby consuming the supply of free pointer-link entries until no more remain. Although having the same processor on the sharing list multiple times does not compromise the correctness of the protocol, lengthy sharing lists take longer to traverse and lead to unnecessary duplicate invalidate requests being issued, substantially increasing protocol processing overhead and degrading system performance.

To resolve this problem, before a processor is added to the sharing list, the list could be checked to see if the processor already is on the list. If so, the processor is not added again. The drawback of this simple solution is that it requires the sharing list to be traversed in its entirety before a new processor is added. The performance impact of the repeated sharing list traversal, especially for large machines, is prohibitive and, as a result, this solution is undesirable in practice.

A better solution is to require the compute processors to inform the home node when a shared line is evicted from the cache. Such notifications are called *replacement hints* and are issued just as writebacks would be, except that replacement hints carry no data since the processor did not modify the cache line. Replacement hints are delivered to the home node, again just as a writeback would be. At the home node, the sharing list is traversed and the processor that issued the replacement hint is removed. Replacement hints eliminate the need to check if a processor already is on the sharing list before adding it and they do so with significantly lower protocol processing overhead than the first proposed solution of checking the sharing list explicitly before a new processor is to be added.

One drawback of replacement hints is that they increase the overall protocol processing overhead because their use requires additional messages to be sent and then processed at the home nodes. However, the performance impact of this drawback is much smaller than the consequences of not using replacement hints [95], and their presence is assumed when using the dynamic pointer allocation protocol on FLASH.[3]

5.3.2.2 Pointer-Link Store Exhaustion

Even with replacement hints, the aggregate size of the sharing lists for all head link entries on a node can lead to exhaustion of free pointer-link entries. The total number of pointer-link entries — that is, the size of the pointer-link store array — is deliberately insufficient to simultaneously handle a large number of sharers for a large number of memory lines on the local node. Directory memory overhead would be unacceptably high if the pointer-link store were sized to accommodate near worst-case sharing patterns without exhaustion. Instead, the pointer-link store is sized according to the expected sharing characteristics and a mechanism is provided to cope with the case in which a new pointer-link entry is needed but none is available.

Lack of a free pointer-link entry is handled by selecting a pointer-link entry at random and then, starting at the selected entry, following the sharing list until the head link is reached. Once the head link is located, the entire sharing list is invalidated just as if a GetX

3. Replacement hints also could be used in conjunction with the bitvector protocol, though not in the coarsevector mode. However, several researchers, such as Weber [38, 108], have found that the advantage of reducing invalidate request and acknowledge traffic by using replacement hints is outweighed by the additional protocol processing overhead required to service the replacement hints. As a result, systems employing the bitvector protocol generally do not use replacement hints; in FLASH, the R10000 processors are configured to disable replacement hint generation when the bitvector/coarsevector protocol is in use.

request had been received. As with the GetX operation described in Section 5.3, upon issuing each invalidation request, the associated pointer-link entry is returned to the free list, thereby making it available to the operation that initially encountered the empty free list.

The process of selecting a pointer-link entry at random, tracing back to the corresponding head link, and then sending invalidate requests to the attached sharing list is called *pointer-link reclamation*. Although reclamation is conceptually straightforward, it incurs a substantial protocol processing overhead, both to generate the invalidate requests and then to process the acknowledgments. In addition, invalidating cache lines selected essentially at random may lead to a performance loss, since the cache lines being invalidated may be in active use by the sharers' compute processors. Overall, a tradeoff must be made in selecting the size of the pointer-link store: if the size is too large, the directory memory overhead will increase and many pointer-link entries may be unused; if the size is too small, pointer-link reclamation will be performed too often and a potentially large performance degradation is likely to result.

5.4 Example Protocol Processing Code Sequences

To make the correspondence among the directory entry formats, protocol operations and the resulting protocol processor code more concrete, this section presents the actual protocol processor code used to service a Get request to a local address in the Clean state for the bitvector protocol. The code implements the home node processing for the simple Get operation flow described in the first part of Section 5.1.2 and illustrated in Figure 5.1. In the nomenclature of the protocol code, the name of the handler for this operation is *NILocalGet* because the message arrives via the NI, targets a local address, and specifies a Get operation. The sections that follow first describe the environment in which protocol code is developed and then present both the C code and protocol processor assembly code for the bitvector NILocalGet handler.

5.4.1 Protocol Code Development Environment

A suite of protocol development tools has been created to allow protocol source code to be written in C and then compiled into PP assembly language. Figure 5.8 depicts the compilation flow from a collection of protocol C source files to a complete protocol code and data image ready for execution on the protocol processor.

Compilation begins by invoking `dpgcc`, a port of the gcc compiler [101] to the protocol processor instruction set, on each C source file. The resulting assembly language (`.s`) files then are processed by a superscalar scheduler called `twine` [98]. `Twine` is config-

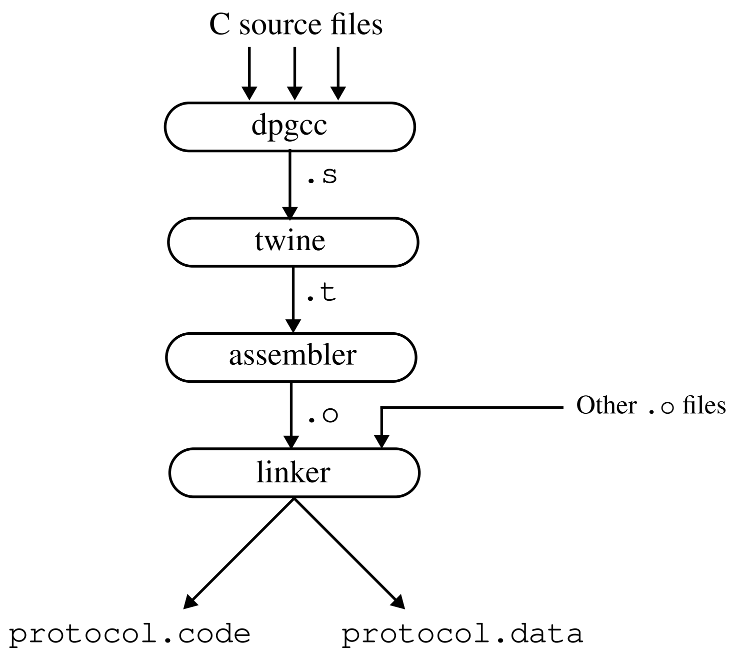


Figure 5.8 Tool flow for protocol processor code development environment

ured with the basic dual-issue capability of the protocol processor as well as various PP scheduling restrictions, such as which instructions can be issued only on one side of the pipeline and the delay slots associated with loads and branches. From twine, a fully-scheduled assembly language file is produced, the `.t` file. Next, each `.t` file is assembled into an object (`.o`) file. Finally, all `.o` files are linked into a single protocol image comprising a `protocol.code` file, containing the protocol code segment, and a `protocol.data` file, containing the protocol data segment. The `protocol.code` and `protocol.data` files are suitable for execution on MAGIC's protocol processor.

5.4.2 Bitvector/Coarsevector NILocalGet Code

Figure 5.9 lists the bitvector NILocalGet C code. This is a portion of a C source file that is supplied to dpgcc for compilation. The code fragment below is meant to provide a concrete example of protocol code; to simplify presentation, the code does not contain all the necessary data structure and macro definitions, and so would require additional header files and support code before it really could be compiled.

Several aspects of the NILocalGet C code are of interest. First, at the C level, protocol handlers are written as if each handler were an individual C function. Normal C data struc-

```
void NILocalGet() {
   DirEntry  *dirEntryAddr, dirEntry;

   /* Compute directory entry address */
   dirEntryAddr = (DirEntry *) ((addr >> 4) + dirBaseAddr);

   /* Load directory entry */
   dirEntry = *dirEntryAddr;

   /* Compose outgoing reply message header, a PUT message */
   header.len = LEN_CACHELINE;
   header.msgType = MSG_PUT;

   /* Start directory state checks */
   if (!dirEntry.Pending)) {
      if (!dirEntry.Dirty)) {
         /* State is Clean/Shared -- send PUT reply to requester */
         NI_SEND(REPLY, F_DATA, F_FREE, F_SWAP, F_NOWAIT);

         /* Add new sharer to vector field */
         dirEntry.Vector |= (1LL << header.src);

         /* Store updated directory entry */
         *dirEntryAddr = dirEntry;
      }
      else {   /* State is Dirty -- more code here */ }
   }
   else { /* State is Pending -- more code here */ }
}
```

Figure 5.9 Bitvector NILocalGet handler C code

tures are used to define the layout of the directory entry and several C preprocessor macros are employed to generate specific PP instructions where needed. The `NI_SEND` macro, for instance, causes the PP *send* instruction to be generated for the purpose of sending the outgoing message header from the PP to the outbox. Second, several variables are initialized automatically: `header` and `addr` are initialized with the incoming MAGIC message header during the PP dispatch operation, and `dirBaseAddr` is a global register that is initialized at system boot with the base address of the directory entry array. Since dispatch and initialization of the message header are managed transparently by the inbox and PP, the result is a very short handler. The only tasks remaining for the handler itself are computing the address of and loading the directory entry, checking the directory state, and composing and sending any necessary outgoing message headers.

```
NILocalGet:
   addi    $5,$0,1          # $5 = constant 1
   srl     $2,$18,4         # Shift incoming address
#--
   add     $24,$19,$2       # Add to directory array base addr in $19
   srl     $8,$17,16        # $8 = source field of incoming header
#--
   addi    $11,$0,8205      # $11 = (LEN_CACHELINE | MSG_PUT)
   ld      $23,0($24)       # Load directory entry into $23
#--
   insfi   $17,$11,0,13     # Create outgoing PUT message header
   sllv    $15,$5,$8        # $15 = (1LL << header.src)
#--
   nop                      # Second load delay slot
   nop
#--
   bbs32   $23,31,$L_Pend   # Branch if state is Pending
   andfi   $27,$23,0,47     # Extract Vector field of dir. entry
#--
   bbs32   $23,30,$L_Dirty  # Branch if state is Dirty
   or      $2,$27,$15       # Add new sharer; $2 is new Vector field
#--
   nop                      # Branch delay slot
   nop
#--
   insfi   $23,$2,0,47      # Update Vector field of dir. entry
   send    $17,$18,15       # Send outgoing message (PUT) to outbox
#--
   switch  $17              # Ask inbox for first half of next header
   sd      $23,0($24)       # Store updated directory entry
#--
   ldctxt  $18              # Ask inbox for second half of next header
   nop
#--
L_Dirty:
        # Code to handle Dirty state here
L_Pend:
        # Code to handle Pending state here
```

Figure 5.10 Bitvector NILocalGet handler PP assembly code

Figure 5.10 shows the same NILocalGet C code after compilation into PP assembly language (dpgcc) and superscalar scheduling (twine). Because the PP is a dual-issue processor, two instructions are executed every cycle; dashed lines are used to separate packets of two instructions in the figure. In addition, the PP is statically scheduled, so explicit *nop* instructions must be inserted where necessary. The resulting assembly code is short — just 9 cycles to the send of the Put reply message and 11 cycles total — and thus demonstrates the need for efficient dispatch and bitfield operations as noted in Section 3.5. Bitfield instructions (*bbs32* and *andfi*, for instance) are used extensively and even a 5-cycle dispatch overhead would increase the handler cycle count by almost 50%. Finally, a close inspection of the C and assembler code demonstrates that the combination of dpgcc and twine leads to efficient, well-scheduled protocol code. Optimizations at the assembly level, such as coding portions of the handlers directly in PP assembly language rather than in C, can yield small improvements in handler cycle counts. But, overall, the benefit of manually coding in assembly language is minimal and is greatly outweighed by the well-known advantages of coding in a high-level language.

5.5 Interaction Between MAGIC and the Operating System

MAGIC is designed primarily to perform protocol processing and thereby implement the system communication paradigm. The MAGIC architecture, particularly the protocol processor instruction set, is designed with this task in mind. In a conventional multiprocessor design, implementing the system communication paradigm usually is all the hardwired node controller can do. Additional functionality is provided only rarely, and in any case must be specified fully at design time rather than after the node controller has been manufactured. With its programmable protocol processor, however, MAGIC is capable of providing more than the communication paradigm; by executing other types of code, MAGIC can provide additional, higher-level services that are traditionally performed by the operating system itself. In this role, MAGIC acts almost as a coprocessor to the R10000, allowing tasks that are more efficiently allocated to the node controller to be performed on MAGIC, as well as allowing MAGIC to invoke OS services to be executed on the R10000.

Key to the implementation of these services is a means to communicate both data and control information between MAGIC and the local R10000. The overall structure of this communication is similar to a remote procedure call: the OS collects the necessary procedure arguments and passes them to MAGIC, along with an indication of what service is to be performed. MAGIC, in turn, executes the requested service and returns a result. MAGIC uses a similar mechanism to invoke OS services on the R10000. Both mechanisms are described below and illustrate some of the additional ways in which a flexible node controller can be used.

5.5.1 OS-to-MAGIC Procedure Calls (PPCs)

Services performed on MAGIC on behalf of the OS are known as protocol procedure calls, or PPCs. The OS invokes a PPC by performing an uncached load to a particular

39–32	31–30	29–18	17–7	6–0
Node	Space	PPC Range	Request ID	Sequence

Uncached load address

Figure 5.11 PPC address encoding

range of addresses that are allocated for PPC use. The particular load address indicates that a PPC is being requested, the exact service to be performed, and the location of any arguments. Figure 5.11 illustrates the encoding of PPC addresses.

The node field specifies the node to which the PPC should be delivered; although remote PPCs are possible, the assumption is that an R10000 will make PPC requests to local address only. The address space field identifies the address as belonging to the I/O space. PPC requests are made as if the MAGIC chip were an I/O device for compatibility with normal OS handling of I/O devices. The PPC range field identifies the address as a PPC rather than a normal I/O PIO operation. The request ID and sequence fields specify the exact PPC operation to be executed as well as the request sequence number.

MAGIC receives the PPC address as an uncached load to the I/O space. The handler invoked on MAGIC must check the PPC range field to determine if the address specifies a PPC or an I/O operation. For a PPC, the handler inspects the request ID to determine how to proceed. Additional arguments to accompany the PPC are supported and are stored in memory starting at the same address as the original PPC load address.

With the PPC request ID and any arguments, the MAGIC handler can process the PPC. After it does so, it responds to the R10000's uncached load with a result code indicating the PPC completion status. In most cases, PPCs are short operations and the MAGIC handler executes them immediately and returns the result. For longer operations, such as those that might require retrieving data from a remote node, the result returned to the R10000 indicates that the request has been accepted and is being serviced. An interrupt is delivered to the R10000 when the operation completes.

Because the PPC mechanism is general, depending only on what code is being run on MAGIC, the set of PPC services is not fixed. Both PPCs originating from the OS itself as well as those requested by applications, with suitable protection provided by the OS, can be accommodated, and the collection of supported PPC services can be adapted to OS and application needs. Table 5.1 lists some of the initial PPC services provided.

Table 5.1 Example PPC services

Service Class	Description
Messaging	Short and long internode message sends and block copies
Config	System configuration control and booting support
Fault	Fault reporting and recovery
Perfmon	Performance monitoring data and configuration; page migration and replication support
I/O	DMA address mapping and I/O device control
Memory	Sharing memory and data structures between OS and MAGIC
Firewall	Fault containment cell control

5.5.2 MAGIC-to-OS Procedure Calls (OSPCs)

Procedure calls can proceed from MAGIC to the R10000 as well. These requests are called OS procedure calls, or OSPCs. MAGIC invokes an OSPC by issuing an interrupt to the R10000, and, to improve service latency, a special interrupt flavor is reserved for OSPC transactions. Because the R10000 does not support "pushes" into its cache (that is, MAGIC cannot force a cache line into the R10000 cache unless the R10000 has requested the line explicitly), a different mechanism is needed to communicate the requested OSPC service and any associated data from MAGIC to the OS. The OSPC argument page is used for this purpose. This is a page in memory to which the OS does not write and does not map into any application address space. In fact, the only operation the OS performs on the OSPC argument page is to read the first cache line. In this way, MAGIC can store the requested OSPC along with any arguments in the first cache line of the OSPC argument page.

The OS begins processing the OSPC interrupt by reading the first line of the OSPC argument page, thereby fetching both the requested OSPC and any arguments into the R10000 cache with the same efficiency as a normal cache miss. After reading the OSPC arguments, the OS flushes the line from the R10000 cache, preparing it for reuse in subsequent OSPCs. Once the OSPC request and arguments have been read, the OS executes the OSPC and returns a result to MAGIC using a PPC reserved for this purpose. Table 5.2 lists some example OSPC services.

As with PPCs, the set of OSPCs is not predefined. In general, most services invoked by a PPC could be performed by the OS itself on the R10000, and some services that MAGIC invokes via an OSPC could be performed on MAGIC instead. As a result, both the set of PPC/OSPC services as well as the allocation of these services to MAGIC and the OS can be tailored to the demands of the OS, application, and MAGIC protocol processing.

Table 5.2 Example OSPC services

Service Class	Description
Error	Report hardware errors and resource overflows
Message	Delivery of internode kernel-to-kernel messages
Interrupt	Delivery of I/O and interprocessor interrupts
TLB	Request virtual-to-physical mapping for protocol use

5.6 Summary

Although development of the MAGIC architecture was guided by the goal of efficient protocol processing, the MAGIC hardware is only half of the solution. Implementing the system's communication paradigm requires both the MAGIC chip itself as well as the protocol code to run on the protocol processor. Whereas the previous chapter focused on the MAGIC architecture and implementation, this chapter has concentrated on two cache coherence protocols: the bitvector/coarsevector protocol and the dynamic pointer allocation protocol. Both protocols share a common set of internode message flows to service cache read misses (Get operations) and write misses (GetX operations). The protocols differ in their directory formats, the mechanisms used to maintain sharing information, and, as a result, in the protocol processing overhead they incur.

The bitvector/coarsevector protocol is optimized for small systems. In its base bitvector usage, the directory format tracks sharers explicitly using a bitvector with one processor per bit. The result is simple, short protocol code sequences and correspondingly low protocol processing overhead. Bitvector is suitable only for small systems, however, because its directory memory overhead grows as the square of the processor count; starting with large systems of perhaps 100 processors, the directory memory overhead becomes prohibitive. The coarsevector modification to the bitvector protocol overcomes the directory memory overhead problem by using one bit in the sharing vector to represent multiple processors, thereby limiting the growth of the sharing bitvector. This modification allows the coarsevector protocol to be used with larger systems but results in a lack of precise sharing information, more complex protocol handlers, and a resulting performance degradation.

By using a linked list of sharers rather than a bitvector, dynamic pointer allocation is able to keep precise sharing information even for large systems. The price is a more complicated directory format, longer protocol handlers, and hence higher protocol processing overhead. Replacement hints also are needed to keep the sharing lists from growing too large because the same processor is added to the list multiple times. Overall, dynamic pointer allocation is best suited to larger systems where the additional overhead of a more complex sharing list and the need to process replacement hints are outweighed by the performance advantages of maintaining precise sharing information.

The presence of the programmable protocol processor in MAGIC offers several advantages. First, it permits the use of a protocol tailored to the system size or even to the anticipated application mix and memory reference characteristics. Small-scale machines are an important component of the multiprocessor market, for example, and a simple, fast protocol such as bitvector is desirable to maximize performance on such systems. MAGIC's flexibility is advantageous in this regard since it allows use of the bitvector protocol for small-scale systems and substitution of a more scalable protocol — such as the dynamic pointer allocation protocol — as the system size grows.

MAGIC's programmability allows other services to be provided in addition to implementing the system communication paradigm. MAGIC acts much like a coprocessor in this role, providing some services that usually would be implemented by the OS on the R10000. The PPC and OSPC mechanisms were developed to allow the OS to invoke services on MAGIC and vice versa.

Chapter 6

System Performance

What is the performance impact of MAGIC's flexible approach to protocol processing? Chapter 3 noted that one goal of FLASH is to use a flexible node controller yet achieve performance competitive with a machine employing the same system architecture but a fully hardwired node controller. This chapter examines whether FLASH achieves this goal.

The performance analysis focuses on two areas. First, the features of the protocol processor instruction set and architecture are examined at a microarchitectural level to quantify their effect on the latency and occupancy of common protocol operations. Second, the performance of FLASH is evaluated at an application level on both parallel scientific codes optimized for a distributed shared memory system and on multiprogrammed workloads, including a full simulation of the operating system. For the scientific codes and the multiprogrammed workloads, performance is measured both in isolation — that is, whether FLASH can achieve good parallel speedups across a range of processor counts — and in comparison to a hardwired machine.

In all cases, the performance analysis deals exclusively with applications that assume a cache-coherent shared memory communication paradigm. FLASH's support for message passing also is important, of course, but the performance of message passing on FLASH is already discussed in detail in [40, 41, 43]. In addition, because an actual FLASH prototype system is still in development at the time of this writing, the performance analysis below is based on a detailed simulation of the FLASH system rather than on data collected from actual FLASH hardware. The FLASH system-level simulator is extremely faithful to the actual hardware design, however, and thus the results reported in this chapter should match closely those that could be gathered on a real FLASH machine.

6.1 Microbenchmarks

To evaluate performance at a low level, the initial performance study measures the number of cycles required for MAGIC to process common protocol operations. This evaluation concentrates on the performance of MAGIC at a microarchitectural level rather than at a system level, and focuses in particular on two figures of merit: the *latency* to service a memory request and the protocol processor *occupancy* incurred in servicing the request. Latency refers to the total number of cycles to service the request, starting when the request is issued on the R10000 SysAD bus and ending when the first word of data is delivered to the R10000. Latency thus measures the total time the R10000 waits for data to be returned from the memory system.

Occupancy refers to the total number of cycles for which the protocol processor is busy servicing a request. Only protocol processor time is included in the occupancy metric; cycles spent traversing the network and in other functional units on the MAGIC chip are excluded. Hence, occupancy provides a measure of the amount of protocol processing overhead a message incurs. The protocol processor also is a point of serialization in the MAGIC architecture, in that the PP can process only one message at a time and additional messages must wait until the PP completes processing the current one. Hence, occupancy also indicates how much serialization, and thus additional queuing delay, the request might cause in the servicing of later messages.

The latency and occupancy data are used to compare MAGIC's protocol operation latencies to those of an actual hardwired node controller. The latency and occupancy information also is gathered for two additional, hypothetical MAGIC designs that use less capable protocol processor architectures, eliminating either the dual-issue capability or the bitfield instruction set extensions. A comparison of the performance of the real MAGIC design to that of the designs with less capable protocol processors quantifies the benefit of the protocol processor's dual-issue and bitfield support.

6.1.1 Simulation Methodology

To gather exact cycle counts for servicing a protocol operation, the Verilog RTL model of the MAGIC chip was used to collect all microbenchmark performance data. Although this is technically a simulation-based approach, the same Verilog RTL used to gather microbenchmark performance data also was used to synthesize the MAGIC netlist for chip fabrication. As a result, the cycle counts measured are by definition the same as in the real hardware, allowing for no inaccuracies resulting from the use of a simulation rather than actual hardware.

6.1.1.1 Simulation Environment

The simulation environment was identical to the one used to perform directed verification on the MAGIC design. In normal operation, this environment would be used to execute a diagnostic test on the MAGIC RTL as part of the overall chip verification process. For the microbenchmarks, however, the diagnostic test was instead used to perform the

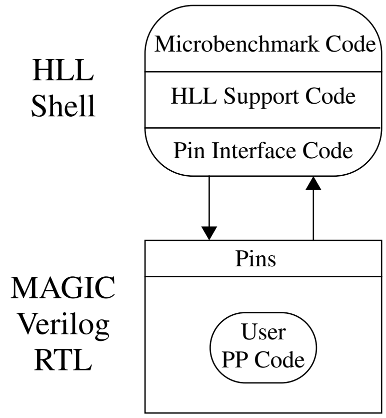


Figure 6.1 Microbenchmark simulation environment

microbenchmark analysis rather than chip verification. Figure 6.1 depicts the high-level structure of the microbenchmark simulation environment.

The environment comprises two parts: the MAGIC Verilog RTL model and a diagnostic shell called the HLL. The RTL models the entire MAGIC chip. As with the real chip, the pins of the MAGIC design form the external interface into the RTL model. The C++-based HLL shell drives the RTL model, controlling and monitoring the MAGIC pins. To use the environment, two pieces of code must be written: one to execute on the protocol processor in the MAGIC chip, and another to execute within the HLL shell. The code running within the HLL determines what transactions are issued to the MAGIC chip and monitors the MAGIC outputs to detect errors. The code running on the protocol processor operates in conjunction with the HLL code to implement the desired diagnostic tests. For the microbenchmark analysis the protocol processor code implements an actual protocol.

6.1.1.2 Microbenchmark Protocol Operations

For the microbenchmarks, the protocol processor code was the actual bitvector protocol routines. The HLL code issued several protocol operations to the MAGIC chip, monitoring the chip outputs and responding as necessary to complete the transactions. Table 6.1 lists the specific protocol operations used in the microbenchmark analysis. These operations cover the five possible read miss (Get) cases in most cache coherence protocols and represent the bulk of protocol processing activity performed for most applications.

Table 6.1 Microbenchmark protocol operations

Number	Read Case	Description
1	Local address, data clean	Read of a local address, data is clean in the node's local memory
2	Local address, data dirty remote	Read of a local address, data is held exclusive in a remote node's compute processor cache
3	Remote address, data clean	Read of a remote address, data is clean in the home node's local memory
4	Remote address, data dirty home	Read of a remote address, data is held exclusive in the home node's compute processor cache
5	Remote address, data dirty remote	Read of a remote address, data is held exclusive in a third node's compute processor cache (three-hop Get)

6.1.1.3 Protocol Processor Variations

As described earlier, one focus of the microbenchmarks is to evaluate the effectiveness of the protocol processor architecture and instruction set. To do so, the five read miss cases were executed three times each, varying the protocol handler code running on the PP to effectively enable or disable the PP's dual-issue and bitfield capabilities.[1] In the first execution, the real protocol processor was used. That is, the handler code made use of both the PP's dual-issue capability as well as the bitfield instruction set extensions, and the performance results from this execution reflect the actual performance of the MAGIC PP.

In the second execution, the handlers were rewritten to continue use of the dual-issue capability but to avoid using any bitfield instructions, in effect modeling a PP that lacks bitfield support in its instruction set. For this case, all bitfield instructions present in the actual handler code were replaced with an equivalent set of shift and logical instructions drawn from the standard MIPS instruction set. The results of this execution measure the benefit of the addition of bitfield support to the PP instruction set.

Finally, for the third execution, the PP handlers were recoded as if the PP were a single-issue processor with no bitfield instructions. Not only were all bitfield instructions in the original handlers replaced in the same manner as for the second execution, but the instruction stream was padded with *nop* instructions such that at most one non-*nop* instruction was executed each cycle. This single-issue, no bitfield experiment gauges the performance of using an off-the-shelf processor code for protocol processing, such as the R3000-based design considered in Chapter 3.

1. "Bitfield capabilities" include both the PP instructions to insert and extract bitfields from a register as well as the branch on bit set/clear instructions.

For all three executions, all protocol handlers exercised by the five read miss operations were coded and optimized by hand directly in PP assembly language; neither the dpgcc compiler nor the twine scheduler was used. Although, as Section 5.4 notes, protocol code normally is coded in C and compiled into PP assembly language, two factors led to the decision to perform manual coding and optimization for the microbenchmarks. First, manual coding eliminates the effect of compiler and scheduler quality. In practice, these tools produce excellent code, but the manually-coded handlers still manage to achieve about a 5–10% instruction count and critical path cycle count reduction versus the compiler-generated code.

Second, and more important, the dpgcc and twine tools are not easily configured to produce code for the second and third executions, that is, those avoiding use of either bitfield instructions or dual-issue capability. As a result, using dpgcc and twine to generate no-bitfield or single-issue code unfairly penalizes the performance of these executions since the quality of the resulting code is significantly lower than the code generated for the actual PP. Hence, manual coding of all handlers for all executions is desirable not only to produce the best overall performance for each execution, but to ensure that the handlers for all executions are equally well coded and optimized.

6.1.2 Latency and Occupancy Results

Figure 6.2 shows the latency results for the five read miss cases for each of the three PP variations. Each read miss case, listed on the X axis, has three associated bars, one for each PP variation as shown in the legend at the right. The latency for each case is expressed in *system cycles*, which is the number of 100 MHz MAGIC clocks required to complete the memory request. Because the clock rate for the R10000 processors is twice the MAGIC clock rate, the latency in processor cycles is simply twice the latency in system cycles.

To focus specifically on the MAGIC architecture, the latency data *excludes* all network transit and R10000 cache intervention time. That is, for cases in which a message must transit the interconnection network from one node to another node or in which data must be retrieved from an R10000 processor cache, the time to perform these activities is excluded from the latency data reported in the figure. The rationale for excluding these delays is that network transit time is a function of the router implementation and network topology, and cache intervention time is a function of the internal details of the R10000, and thus neither of these delays is related the MAGIC architecture.

Table 6.2 lists the latency differences for each read miss case; the numbers in the read case column refer to those in Table 6.1. Overall, the actual PP provides an 8–20% latency

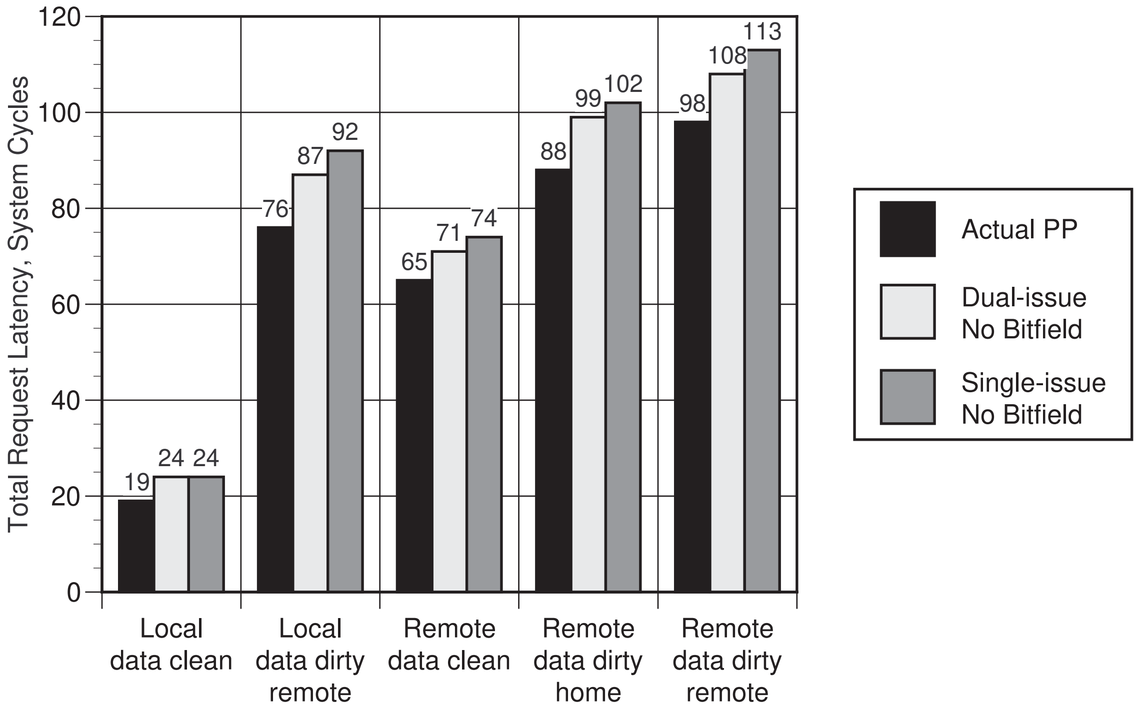


Figure 6.2 Microbenchmark read miss latencies

reduction versus the dual-issue, no bitfield PP and a 12–20% latency reduction versus the single-issue, no bitfield PP. The latency advantage of the real PP is particularly crucial for the local read case (case 1), in which the lack of bitfield instructions increases the number of instructions required to test the directory state, leading to an additional five-cycle latency before the data is returned.

Table 6.2 Microbenchmark latency reductions for actual PP

Read Case	Latency reduction versus dual-issue, no bitfield PP	Latency reduction versus single-issue, no bitfield PP
1	20.8%	20.8%
2	12.6%	17.4%
3	8.5%	12.2%
4	11.1%	13.7%
5	9.3%	13.3%

Although the advantages of the actual PP are evident in the latency data, the occupancy data demonstrate a more dramatic improvement, as shown in Figure 6.3. Like the latency data, the occupancy data exclude cache intervention delays; network transit time is not a factor in occupancy measurements since only the time the PP spends processing the message header is measured. For all cases except the local read to a clean line (case 1), PP occupancy is incurred more than once during the course of the operation. A remote read to a clean line (case 3), for instance, starts by invoking the PP at the requesting node to receive the initial read and forward it to the home node; then the PP at the home node is invoked to retrieve the data from memory and send it to the requester; finally the requester's PP is invoked once again to receive the data from the home and return it to the local R10000. In these cases, the occupancy reported is the sum of the occupancies incurred by all protocol processors involved in servicing the request.

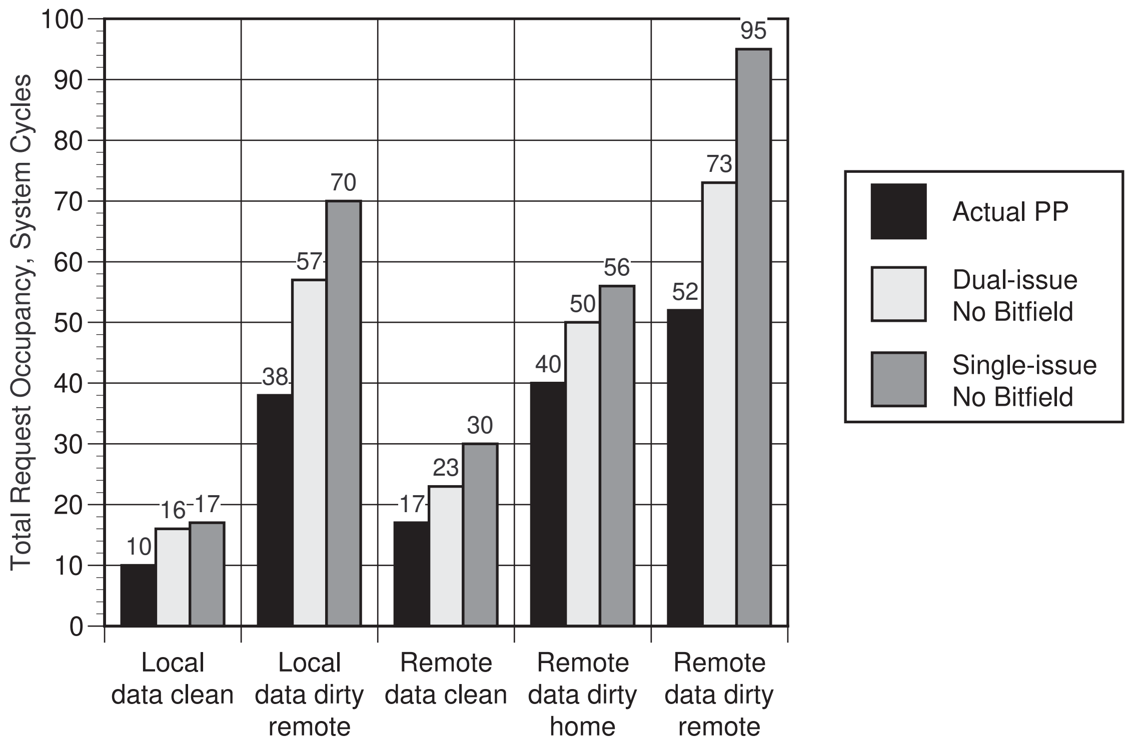


Figure 6.3 Microbenchmark read miss occupancies

The occupancy results show that the real PP achieves a 20–37% occupancy reduction versus the dual-issue, no bitfield variation and a 28–45% reduction versus the single-issue, no bitfield variation; Table 6.3 compares the PP variations for all five read cases. The magnitude of the occupancy reductions demonstrates the utility of the bitfield and dual-issue features, particularly since occupancy is a good indication of overall protocol processing overhead.

Table 6.3 Microbenchmark occupancy reductions for actual PP

Read Case	Occupancy reduction versus dual-issue, no bitfield PP	Occupancy reduction versus single-issue, no bitfield PP
1	37.5%	41.2%
2	33.3%	45.7%
3	26.1%	43.3%
4	20.0%	28.6%
5	28.8%	45.3%

6.1.3 Analysis of Latency and Occupancy Data

One difficulty with both the latency and occupancy measurements is that they do not translate directly into system performance. A system having a slower node controller with double the latencies and occupancies of MAGIC will not necessarily have worse overall system performance. By the same token, a node controller with lower latencies and occupancies will not always yield superior overall system performance. Nevertheless, the microbenchmark analysis presented above is useful in two ways: to quantify the benefit of the protocol processor dual-issue and bitfield features and as a gauge of how favorably MAGIC's latency and occupancy figures compare to those of a system using a fully hardwired node controller.

6.1.3.1 Benefit of Dual-Issue and Bitfield Support

Adding dual-issue and bitfield capability to the protocol processor increased the design effort and added measurably to the die area occupied by the PP. The dual-issue feature increased design complexity because it required two execution pipelines rather than one. Even with static scheduling, coordinating and sequencing the two pipelines added logic complexity and verification time. The extra execution pipeline also increased die area, as did the need to add additional bitfield-specific functional units and the need to support bypassing within each execution pipeline as well as between the two pipelines. In addition, neither feature is found in a typical off-the-shelf processor core.

Consequently, the dual-issue and bitfield support must result in significantly lower latencies and occupancies if the additional design effort, die area, and inability to use an off-the-shelf core are to be justified. As Table 6.2 and Table 6.3 demonstrate, these features do result in significantly lower latencies and, especially, occupancies. The latency advantages are relatively modest (about 10%) except on the local read to a clean line. Local, clean reads often are the most frequent operation for many applications, however, and the 20% latency advantage in this case is thus extremely desirable.

The occupancy advantages of the real PP, even when compared to the dual-issue, no bitfield variation, are clear for all five read miss cases. Unfortunately, the connection between occupancy and performance is not intuitive. However, as Holt [50] and others [24, 71, 74, 78] have concluded, node controller occupancy has a substantial impact on system performance. Taken as a whole, the latency and occupancy advantages of the real PP justify the additional effort required to add dual-issue and bitfield support.

6.1.3.2 Comparison to a Hardwired Node Controller

The latency data also can be analyzed by comparing FLASH latencies to those of a real hardwired machine. Table 6.4 presents the latency data for FLASH and for the SGI Origin 2000 [51], a hardwired system. Again, network transit and R10000 cache intervention time have been excluded from these latency values. The MAGIC chip and the Origin 2000 node controller have the same target clock rate and similar ASIC implementation technologies, so the two designs faced similar implementation constraints, increasing the validity of the latency comparison.

Table 6.4 FLASH and Origin 2000 read miss latencies

Read Case	FLASH	Origin 2000	Origin Speedup
1	19	20	–5.0%
2	76	45	68.9%
3	65	41	58.5%
4	88	48	83.3%
5	98	59	66.1%

As the table shows, the FLASH latencies are approximately 60–80% higher than the Origin latencies, with the exception of the local read of a clean line (case 1) for which the latencies are essentially equal. However, as demonstrated by the application performance results section that follows, raw latency differences do not necessarily lead to the same difference in overall application performance since the latency differences impact performance only when a processor is stalled on the memory system. Section 6.3 investigates this issue in greater detail.

6.2 Application Simulation Methodology

The microbenchmark data presented in Section 6.1 characterize MAGIC at a low level. Ultimately, however, the specific latency and occupancy of individual read miss cases are only one, albeit the major component of overall system performance. To accurately characterize the MAGIC and FLASH architecture, the performance also must be examined at a system level, using actual applications to measure the impact of MAGIC's flexible approach to protocol processing. This and the following two sections delve into system performance in detail. This section explains the simulation methodology used for the system performance evaluation. Next, Section 6.3 presents the performance results of several scientific applications that have been written specifically for use on a distributed shared memory machine. Last, Section 6.4 examines a different aspect of FLASH performance by considering multiprogrammed workloads, including a full simulation of the operating system, which is a significant factor in the performance of these applications.

6.2.1 Parallel Scientific Application Set and Simulation Environment

All of the parallel scientific applications were drawn from the SPLASH-2 benchmark suite [110, 111] and are representative of a range of scientific computational methods. Each application is coded in C and augmented with the Argonne National Laboratory (ANL) parallel macro package [14] to provide global memory allocation, synchronization, and other parallel shared memory programming primitives. Table 6.5 lists the five applications selected from the SPLASH-2 suite, along with a brief description of what class of scientific computation each application represents, as well as the problem size used. Section 6.3 describes each application in detail.

Table 6.5 Scientific applications and problem sizes

Application	Representative Of	Problem Size
Barnes	Hierarchical N-body codes	8192 particles; $\theta = 1.0$
FFT	Transform methods; high-radix	256K complex points; radix $\sqrt{N}$
LU	Blocked dense linear algebra	768-by-768 matrix; 16-by-16 blocks
Ocean	Regular-grid iterative codes	258-by-258 grids
Radix	High-performance parallel sorting	1M/2M integer keys; radix 256

Problem size selection often is problematic for scientific applications since the codes are suitable for a variety of problem sizes, and certain undesirable machine performance traits can be unfairly masked by using overly large problem sizes [89, 96]. For this study, the problem sizes were selected to yield at least 0.60 parallel efficiency[2] at 64 processors on a machine with a hardwired node controller [49, 50]. This criterion results in realistic problem sizes for both large and small processor counts and ensures that the problem sizes

2. For a given processor count P and an application speedup of S at that processor count, parallel efficiency is defined as (S / P). Hence, a parallel efficiency of 1.0 indicates linear speedup at the given processor count.

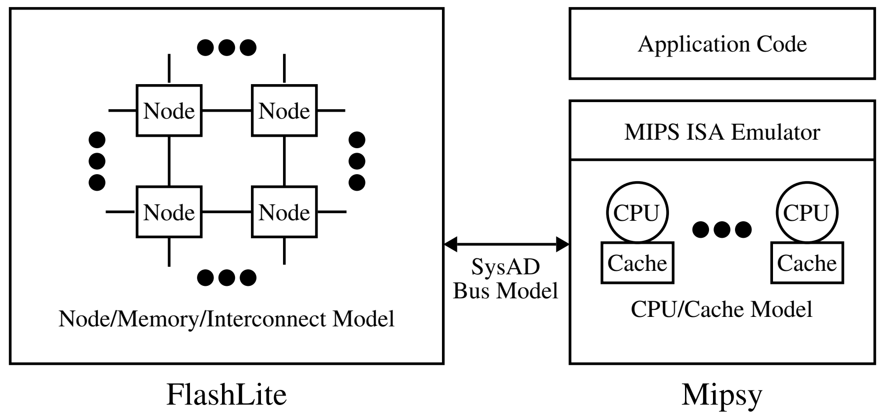


Figure 6.4 Scientific application simulation environment

are tailored to a hardwired machine, thereby helping to eliminate the possibility that performance problems in the FLASH system will be masked by problem sizes that are too large.

To properly simulate the SPLASH-2 applications, a simulator must support both a large range of processor counts as well as a detailed model of the memory system. Because these applications make almost no system calls and, even on a real machine, spend very little time in the operating system [26, 27, 37], a detailed model of the OS is not necessary to model SPLASH application performance accurately. The simulation environment developed to support these requirements is depicted in Figure 6.4.

The environment comprises three basic components: the Mipsy CPU simulator, the FlashLite memory system simulator, and the application code. Together, these components permit a detailed simulation of the FLASH system driven by execution of an actual application. The Mipsy CPU simulator is responsible for simulating all R10000 compute processors and their first-level and second-level caches. For an N-processor simulation, Mipsy instantiates N processor/cache models, each executing in parallel. A multithreaded simulation core underlies the Mipsy processor and cache models, allowing the concurrent operation of the N processors to be modeled accurately, including all contention and queuing effects.

Mipsy models only the processors and their local caches. To model a complete multiprocessor, a memory system simulator is needed, and this role is performed by FlashLite. As shown in figure, FlashLite models everything in a FLASH system except for the compute processors and caches. Hence, FlashLite models the MAGIC chip, the local memory on each node board, and the interconnection network that connects the nodes. Like Mipsy, FlashLite supports N-node simulations using a multithreaded core to allow modeling of the concurrent operation of N node boards and the routers in the interconnection network.

FlashLite is designed specifically to model FLASH, and contains extremely detailed models of the Spider routers as well as the internal structure and logic in the MAGIC chip. The protocol processor is modeled in great detail, and the same protocol code compiled for use on an actual FLASH machine is used with FlashLite. As a result, both the system hardware characteristics as well as the protocol code are modeled faithfully and the simulated performance of the FLASH system should match closely the actual performance of a real FLASH machine.

Mipsy and FlashLite together model an entire FLASH system. All that is needed is an application to drive the simulation system. As part of its processor model, Mipsy contains a MIPS instruction set emulator. Thus, as the figure shows, an application can be compiled and linked exactly as if it were going to be run on a a real FLASH machine. Instead, however, the application executable image is supplied as input to Mipsy. Mipsy emulates the application instructions and tracks the state of the processor caches. When a second-level cache miss is detected, Mipsy issues the proper request to FlashLite — the interface between Mipsy and FlashLite is a model of the R10000 SysAD bus. FlashLite in turn models the memory system operations necessary to satisfy the memory request. After the operation completes, FlashLite signals the completion and returns the associated data to Mipsy, satisfying the cache miss. The multithreaded core underlying both Mipsy and FlashLite tracks the number of cycles required for instruction emulation and memory system activity, allowing accurate performance data to be collected.

For the scientific application results presented in Section 6.3, Mipsy was configured to model a 1 MB R10000 secondary cache and to assume that the R10000 could issue as many as three instructions per system cycle. That is, Mipsy effectively models a single-issue processor running at 300 MHz, or three times the MAGIC (system) clock rate. In addition, all the applications were simulated using the bitvector/coarsevector protocol described in Section 5.2. For processor counts under 48, the bitvector variation was employed; for the 64-CPU runs, coarsevector was used with a coarseness factor of two.

6.2.2 Multiprogrammed Workload Set and Simulation Environment

Scientific applications represent one realm of usage for a multiprocessor. Also important is performance on multiprogrammed workloads in which the primary emphasis is on using the multiprocessor's ability to efficiently share resources such as processors, memory, and I/O devices. Whereas the parallel applications focus on applying a large number of processors concurrently to speed the solution of a single job, the focus in the multiprogrammed workloads is on throughput. That is, many individual jobs are executed and the goal is to complete them all as quickly as possible.

The multiprogrammed workload set comprises two workloads, summarized in Table 6.6. The table lists the name of each workload, what type of computation it represents, and an overview of the input data characteristics. Section 6.4 describes the workloads in detail and presents performance results. In general, the workloads differ from the scientific applications in that they are multiprogrammed rather than large-scale parallel, and in that they are not tuned for a distributed shared memory system. The simulation also includes the operating system.

Table 6.6 Multiprogrammed workload set

Workload	Representative Of	Problem Size
PMake	OS-intensive compilation	8-way make of gnuchess; total of 9426 lines of C
Engineering	Large working sets	Combination of FlashLite and VCS

Because the multiprogrammed workloads consist of multiple jobs executing concurrently, their execution requires the ability to spawn multiple independent processes across the machine, schedule the processes, and manage their interaction with each other and with any I/O devices they access. These requirements differ substantially from the needs of the scientific applications, for which, in general, only the ability to execute the same code on multiple CPUs in parallel is required. In particular, the scientific applications performed no I/O and assumed that they were the only application executing on the system; no process scheduling was used. A consequence of the more demanding requirements of the multiprogrammed workloads is that the operating system no longer can be ignored. The applications make many system calls and rely on the OS to perform numerous services on their behalf. To generate accurate performance data for these applications, then, both the application code as well as the OS must be simulated.

Simulation of the operating system is a challenging problem. Although most of the OS code can be run in user mode, the portions of the OS that use privileged instructions and assume direct access to the underlying system hardware must be executed in supervisor mode. As a result, simulation of the OS requires not only a mechanism for emulating the functionality of the privileged instructions but also a complete model of the system hardware. The SimOS framework [86, 88] achieves both these goals and allows simulation of a complete machine, including actual application code, the actual, unmodified OS code, and realistic models of system hardware devices such as disk drives, ethernet controllers, and

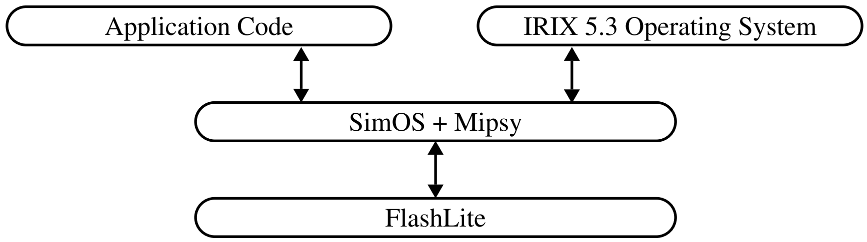


Figure 6.5 SimOS-based multiprogrammed workload simulation environment

the like. Figure 6.5 shows how the SimOS framework is integrated with the Mipsy and FlashLite components to provide a comprehensive simulation environment for the multiprogrammed workloads.

The SimOS environment is very similar to the scientific application simulation environment described in Section 6.2.1. The major change is the addition of the SimOS component and the actual operating system image. Mipsy still models the compute processor and its caches, but rather than emulating just the application code, Mipsy now emulates both the operating system code and application code. Privileged instructions and direct access to hardware devices are trapped in Mipsy and passed to SimOS for emulation. Second-level cache misses are issued to FlashLite just as in the scientific application environment, and the same multithreaded core underlies all components and allows accurate tracking of cycle counts, latencies, contention, and other performance information.

From FlashLite's perspective, the SimOS environment is no different from the scientific application environment, except that execution of the operating system causes a greater variety of processor references to be issued. With SimOS, for example, FlashLite will receive not only normal cache read and write misses, but uncached operations as well. Some uncached operations are actually PIO operations targeted at the I/O subsystem. As a result, use of SimOS requires a complete protocol, including handlers for I/O operations.

As with the scientific application results, for the multiprogrammed workloads Mipsy was configured to model a 1 MB R10000 secondary cache and to assume that the R10000 could issue as many as three instructions per system cycle. For the multiprogrammed workloads, however, the dynamic pointer allocation protocol was used rather than bitvector/coarsevector.

6.2.3 Three Simulated Machines

For both the scientific codes and the multiprogrammed workloads, the goal is to evaluate FLASH system performance in isolation and compared to a machine that retains the basic FLASH system architecture but replaces the PP with a fully-hardwired protocol engine. To do so, the FlashLite simulator was configured to model three different machines: FLASH, the *ideal machine*, and the *hardwired machine*.

In the FLASH configuration, FlashLite models the actual FLASH system as closely as possible, including both the internal functional units in MAGIC and the routers that form the interconnection network. Actual protocol code is used as well, so that the same protocol handlers that would be executed on the real FLASH system also are used in FlashLite's model of FLASH.

The ideal machine retains the same system architecture as the FLASH machine — that is, the same internal MAGIC architecture and interconnection network — but replaces the real protocol processor with an idealized version that executes all protocol handlers in zero time. In other words, when a MAGIC chip on the ideal machine receives a message, the message is processed by the source interface unit and inbox in the same way as in the real FLASH system. Once the message header reaches the PP, however, the associated handler executes in zero cycles. This means that the directory state is read, tested, and updated, and any required outgoing messages are composed and sent, all with no delay. Both the MAGIC instruction and data caches are assumed to be perfect (100% hit rate), and the only delays the ideal PP incurs are those to perform interventions in the local R10000 compute processor cache. In addition, the latencies of the various interface units, as well as the inbox and outbox, are reduced for the ideal machine. The rationale behind the ideal machine is not to model a realizable system; obviously, a zero-delay PP could not be built. Rather, the goal is to set an upper bound on the performance achievable by the basic FLASH and MAGIC architecture while intentionally ignoring the effects of protocol processing overhead.

Like the ideal machine, the hardwired machine retains the same FLASH system and MAGIC architecture. But, instead of replacing the PP with a zero-delay model, the hardwired machine's PP has occupancies based on those of the SGI Origin 2000 node controller (the "Hub" chip), an actual, commercially-available hardwired system. That is, when the hardwired machine's PP services a message, the regular protocol code is executed, but the handler execution cycle count is assumed to be the same as the SGI Hub chip requires to service the same message. In most cases, the Hub chip requires two cycles of occupancy for each message header. More complex message processing, in particular messages that require invalidate requests to be sent, incur extra occupancy beyond the two-cycle base.

The goal of the hardwired machine is *not* to model the performance of the Origin 2000. The Origin 2000 has a different system architecture and different node controller architecture, and the performance results of the hardwired machine should not be expected, and are not intended, to represent the performance of the Origin 2000. Instead, the goal is to model a machine that retains the FLASH and MAGIC architecture but uses a hardwired protocol processor. This combination provides a reference for how much performance loss the actual FLASH system incurs in using a flexible, programmable protocol processor. By using the occupancies of the Hub chip, additional simulation realism and accuracy is achieved because the hardwired machine's occupancies are based on a real hardwired node controller design that faced design constraints comparable to the MAGIC chip — the same target clock rate and similar ASIC implementation technology, for example.

6.3 Parallel Scientific Application Performance

Three performance metrics are of interest for the scientific applications: self-relative speedup, FLASH-relative speedup, and the relative contribution of protocol processor latency and occupancy. Self-relative speedup refers to a speedup calculation that compares the execution time of a given machine (FLASH, hardwired, or ideal) at a processor count P to the execution time of the same machine for a uniprocessor run. Thus, self-relative speedup is a measure of whether the machine is scalable; that is, whether it can deliver good speedups across a range of processor counts.

FLASH-relative speedup is similar to self-relative speedup, except that FLASH-relative speedup compares the execution time of a given machine at a processor count P to the FLASH execution time at the same processor count. Thus, FLASH-relative speedup measures, for a particular processor count, how much faster the hardwired or ideal machine is compared to FLASH.

The relative contribution of latency and occupancy refers to how much performance loss can be attributed to FLASH's programmable protocol processor versus the hardwired machine's hardwired PP. Not only is the absolute execution time difference between FLASH and the hardwired machine of interest, so too is the relative contributions of the additional PP latency and occupancy. To differentiate latency effects from occupancy effects, a measure called the *contentionless read miss latency* (CRML) is used. CRML indicates how many system cycles each of the three machines requires to service the five read miss cases noted in Table 6.1. The CRML statistics for the three machines for a system size of 32 CPUs — the size used for the CRML analysis in the sections that follow — are shown in Table 6.7.

Table 6.7 Contentionless read miss latencies, system cycles

Read Case	32 Processors[a]		
	FLASH	**Hardwired**	**Ideal**
Local, data clean (case 1)	21	21	18
Local, data dirty remote (case 2)	117	96	85
Remote, data clean at home (case 3)	94	89	82
Remote, data dirty ay home (case 4)	134	97	86
Remote, data dirty remote (case 5)	159	117	114

a. Latencies include, where appropriate, R10000 cache intervention time and an average network transit delay of 31 cycles (average of 3.1 routers traversed in a 32-CPU, 4-D hypercube).

Although the CRML is listed for all three machines, the most meaningful comparison is between the FLASH and hardwired machines; the ideal machine is meant to set an upper bound on performance rather than to be representative of a real system. By multiplying the actual read miss distribution observed in an application run by the CRML cycle counts, an effective CRML can be calculated for the FLASH and hardwired machines. If the ratio of the two effective CRML values then is multiplied by the fraction of time the hardwired machine is stalled in the memory system, the result indicates how much the *latency* difference would increase this stall time for FLASH. Any additional execution time difference between FLASH and the hardwired machine is due to *occupancy* effects. In this way, the CRML, read miss distribution, and memory system stall time can be used to separate the performance difference between the FLASH and hardwired machines into latency and occupancy components.

The five sections that follow cover each scientific application. Each section begins with a brief description of the application and then reports its self-relative speedup, FLASH-relative speedup, and several other performance metrics for the FLASH machine. The CRML analysis presented above is performed as well. The application descriptions are intentionally brief; a full discussion of each application can be found in Woo [109, 110].

6.3.1 Barnes

Barnes is an N-body simulator for modeling the interaction among a system of particles in three-dimensional space over a series of time steps. The application uses the Barnes-Hut hierarchical N-body algorithm to compute the forces among the particles [48]. The three-dimensional particle system is represented using an octree data structure. Each time step involves traversing the octree and computing the forces among the particles, a process that leads to unstructured communication, typically of small data items such as the octree leaves used to store information about the particles.

The top of Figure 6.6 shows the self-relative speedups for all three machines. The number of processors, shown on the X axis, varies from 1 to 64 and the self-relative speedup at each processor count is shown on the Y axis. Another way to view the speedup data is by computing the FLASH-relative speedups, which are shown at the bottom of Figure 6.6. As is the case for all the scientific applications, both the self-relative and FLASH-relative speedup graphs demonstrate that FLASH achieves competitive speedups in isolation and competitive performance when compared to the hardwired machine. Only at 64 processors does FLASH's performance begin to lag the hardwired machine's; at 64 CPUs, the hardwired machine is approximately 10% faster than FLASH on Barnes.

Table 6.8 lists a number of important execution characteristics of Barnes on FLASH. The table is divided into four sections that focus on the performance of the application as a whole and on its impact on the protocol processor. The meaning of the fields in the leftmost column is as follows:

- *Speedup* — Self-relative speedup versus the uniprocessor execution time.
- *Total execution time* — Expressed in millions of system cycles.
- *Processor utilization* — Fraction of time the compute processors are busy, averaged across all processors.
- *Synchronization stall time* — Fraction of time the compute processors are stalled waiting for synchronization operations to complete.
- *Cache reads, cache writes* — Fraction of compute processor cache accesses that were reads and writes.
- *Cache miss rate* — Miss rate of the compute processor's secondary cache, referenced to all processor loads and stores, not just to those that miss in the primary cache.
- *Reads, local clean* through *Reads, remote, dirty remote* — Division of all cache read misses into the five cases described in Table 6.1.
- *Average read miss latency, Average write miss latency* — Number of system cycles a read or write took to complete, averaged across all processors.

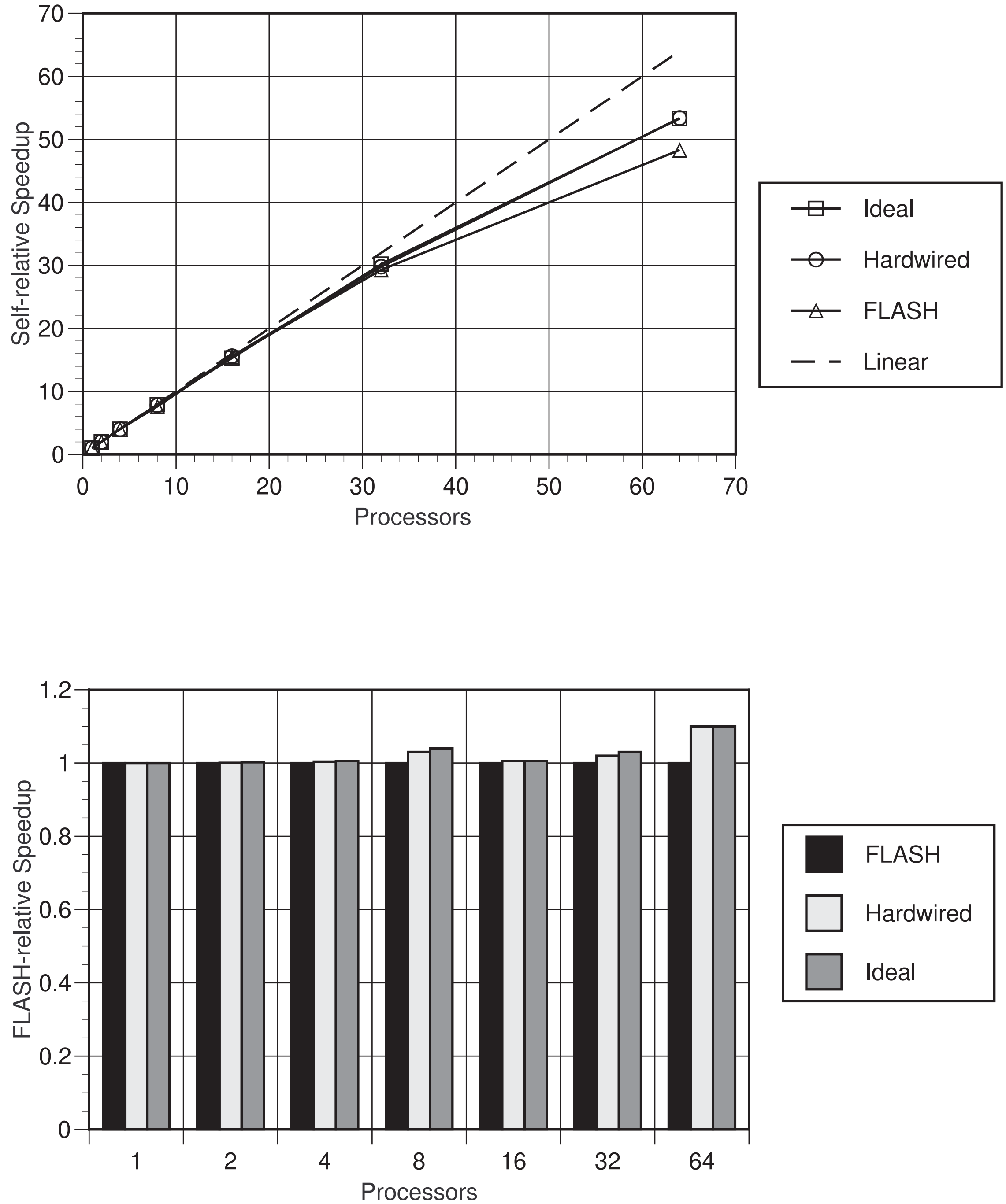


Figure 6.6 Barnes self-relative (top) and FLASH-relative (bottom) speedups

Table 6.8 Barnes execution characteristics on FLASH

Execution Characteristic	Processor Count						
	1	2	4	8	16	32	64
Speedup	1.0	2.0	4.0	7.6	15.5	29.3	48.3
Total exec. time (millions)	368.28	185.20	92.74	48.37	23.77	12.59	7.63
Processor util. (%)	99.4	98.6	98.5	94.5	96.3	90.9	74.6
Sync. stall (%)	0.0	0.1	0.4	2.1	2.3	6.1	19.5
Cache reads (%)	67.4	67.4	67.5	68.1	68.1	68.8	71.7
Cache writes (%)	32.6	32.6	32.5	31.9	31.9	31.2	28.3
Cache miss rate (%)	0.01	0.02	0.01	0.02	0.01	0.01	0.02
Rds, lcl, clean (%)	100.0	72.9	47.7	15.9	7.8	5.6	4.7
Rds, lcl, d. rem (%)	0.0	0.1	0.2	0.2	1.0	0.7	0.8
Rds, rem, clean (%)	0.0	25.5	41.7	77.0	62.6	78.8	83.1
Rds, rem, d. home (%)	0.0	1.5	10.1	6.5	25.7	12.9	10.4
Rds, rem, d. rem (%)	0.0	0.0	0.2	0.4	3.4	2.4	1.9
Avg. read miss lat.	26	37	54	72	98	101	155
Avg. write miss lat.	29	30	36	55	76	91	181
Avg./Max. PP util. (%)	0.6/0.6	0.7/0.8	0.7/0.8	1.2/2.4	1.0/1.4	1.5/4.4	2.8/10.6
Dynamic dual-issue eff.	1.38	1.42	1.40	1.42	1.38	1.38	1.30
Special instr. fraction	0.49	0.51	0.50	0.53	0.50	0.50	0.43
MIC miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.10
MDC miss rate (%)	0.10	0.00	0.00	0.00	0.00	0.00	0.00

- *Average/Maximum PP utilization* — Average and maximum fraction of cycles for which a PP was busy processing a message header.
- *Dynamic dual-issue efficiency* — Average number of non-*nop* instructions the PP executes per busy cycle. Perfect use of the PP's dual-issue capability would yield a dynamic dual-issue efficiency of 2.
- *Special instruction fraction* — Dynamic fraction of all non-*nop* PP instructions that are bitfield, branch on bit, or functional unit interface instructions.
- *MIC miss rate, MDC miss rate* — Miss rate of the MAGIC instruction and data caches, respectively.

The R10000 cache miss rate explains why FLASH yields competitive performance despite the latency disadvantages noted in Table 6.7. Because the miss rate is very low, the program spends very little time in the memory system. As a result, the larger latencies of the FLASH machine do not lead to much difference in overall execution time; essentially, the latency difference is masked by the minimal amount of time spent in the memory system. This trend pervades the scientific application performance results.

At 64 CPUs, FLASH's performance does fall off compared to the hardwired machine's. Two factors contribute to FLASH's relatively worse performance. First, synchronization time is no longer negligible (as it was for 32 and fewer CPUs) and synchronization across the 64 processors is handled faster by the hardwired machine. Second, the coarsevector protocol must be used at 64 CPUs. The additional protocol processing and message overhead of coarsevector versus bitvector affects FLASH's performance more than the performance of the hardwired machine. Section 6.3.3 explores this point further.

A CRML-based analysis, as discussed at the start of Section 6.3, can be employed to separate the performance degradation of FLASH versus the hardwired machine into latency and occupancy components. This computation makes most sense to perform at 32 CPUs since the performance of the two machines is almost identical at smaller CPU counts, and at 64 CPUs other effects begin to dominate.[3] Table 6.9 summarizes the 32-processor CRML results.

Table 6.9 Barnes 32-processor CRML calculation

Characteristic	FLASH	Hardwired
Memory stall time (%)	2.09	
Execution time (sys. cycles x10^6)	12.59	12.37
Effective CRML (sys. cycles)	97.05	87.60
Hardwired speedup (%)	1.76	N/A
Latency component (%)	100.0	N/A
Occupancy component (%)	0.00	N/A

The CRML calculation indicates that all of the execution time difference between FLASH and the hardwired machine can be attributed to latency effects. This result is explained by the unstructured communication patterns and lack of prefetching in Barnes, which expose the additional latency of the FLASH memory system, and by the overall good processor cache hit rate, which means that hot-spotting is not likely to occur, thus minimizing the impact of occupancy.

3. At 64 processors, the CRML results are either similar or show more of an impact from occupancy effects, a consequence of the increase in synchronization stall time at 64 CPUs. Contention at the home nodes of the synchronization variables exacerbates the occupancy effects as well.

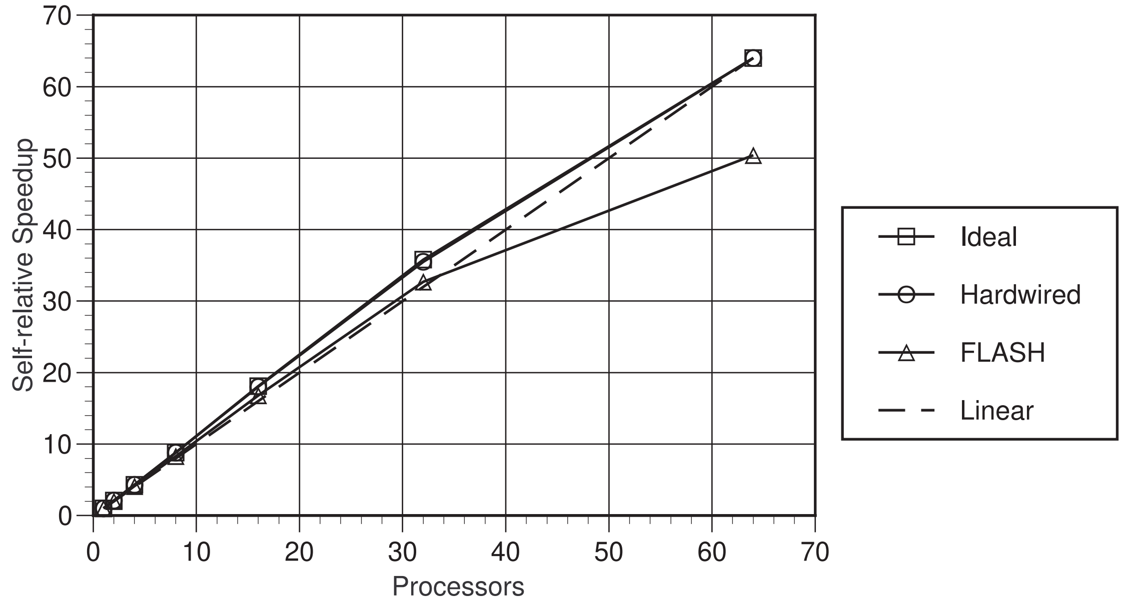
6.3.2 FFT

The FFT kernel performs a one-dimensional complex fast Fourier transform, using the six-step radix-$\sqrt{n}$ algorithm described in Bailey [9]. Two $\sqrt{n}$-by-$\sqrt{n}$ matrices are used, one containing the data to be transformed and another containing the roots of unity. Communication among the processing nodes in FFT occurs during each of three matrix transpose steps. Unlike in Barnes, communication in FFT is very structured and requires all-to-all communication rather than the more random pattern observed in Barnes. In addition, the communication activity is staggered to minimize contention in the interconnection network and the matrix transpose operations are blocked to reduce communication and maximize processor cache hit rate. In all, FFT is extremely well optimized for a distributed shared memory environment.

Figure 6.7 shows the self-relative and FLASH-relative performance for FFT. Both prefetched and non-prefetched versions of the algorithm were simulated; the prefetched version achieves approximately 10% better overall performance and would be used in practice, so only prefetched FFT results are shown in the figure. Table 6.10 lists the FFT execution characteristics on FLASH.

Continuing the theme observed with Barnes, the FFT results demonstrate that FLASH achieves good speedup for all processor counts and overall performance competitive with the hardwired machine at all but 64 processors. As with Barnes, the low processor cache miss rates mean that little time is spent in the memory system, helping to minimize the execution time difference. At 64 CPUs, both the synchronization overhead and the impact of the transition into a coarsevector protocol affect FLASH's performance.

Table 6.11 lists the 32-processor CRML results. To emphasize the effect of prefetching on the CRML results, the table shows the CRML calculation for both prefetched and non-prefetched executions. The prefetched CRML result indicates that most of FLASH's performance loss is due to occupancy effects whereas the non-prefetched result indicates that most of the performance loss is due to latency effects. This confirms the ability of the prefetch operations to hide memory system latency, at least in FFT, and highlights the criticality of occupancy when not hiding latency. With prefetches in use, not only is the overall performance of both machines better, but the performance difference between FLASH and the hardwired machine becomes a result of occupancy rather than latency.



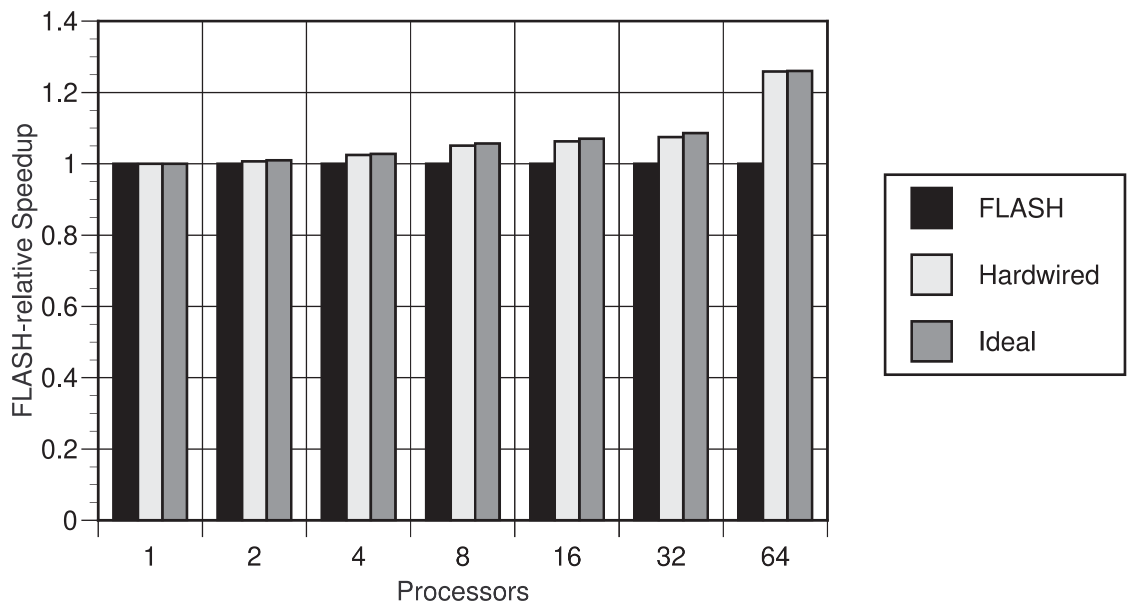


Figure 6.7 FFT self-relative (top) and FLASH-relative (bottom) speedups

Table 6.10 FFT execution characteristics on FLASH

Execution Characteristic	Processor Count						
	1	2	4	8	16	32	64
Speedup	1.0	2.0	4.1	8.3	16.8	32.7	50.4
Total exec. time (millions)	70.34	34.72	17.12	8.51	4.20	2.15	1.39
Processor util. (%)	92.6	94.3	95.6	96.3	97.8	97.0	88.7
Sync. stall (%)	0.0	0.0	0.4	1.4	1.4	2.3	8.7
Cache reads (%)	59.2	59.0	59.1	59.4	59.3	59.6	62.0
Cache writes (%)	40.8	40.9	40.8	40.6	40.6	40.4	37.9
Cache miss rate (%)	0.29	0.29	0.25	0.20	0.16	0.15	0.15
Rds, lcl, clean (%)	100.0	74.9	57.6	37.6	13.7	2.6	2.2
Rds, lcl, d. rem (%)	0.0	0.0	0.0	0.1	0.1	0.3	0.6
Rds, rem, clean (%)	0.0	24.3	35.2	30.2	30.5	33.6	34.0
Rds, rem, d. home (%)	0.0	0.8	7.2	32.2	55.7	63.4	63.5
Rds, rem, d. rem (%)	0.0	0.0	0.0	0.0	0.0	0.1	0.1
Avg. read miss lat.	26	26	27	28	35	88	203
Avg. write miss lat.	33	40	44	51	53	49	97
Avg./Max. PP util. (%)	9.1/9.1	11.8/11.8	12.7/12.9	13.0/13.4	13.1/13.3	13.3/13.8	22.1/23.7
Dynamic dual-issue eff.	1.40	1.40	1.40	1.42	1.42	1.42	1.30
Special instr. fraction	0.48	0.49	0.49	0.49	0.49	0.49	0.41
MIC miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MDC miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 6.11 FFT 32-processor CRML calculation

Characteristic	Prefetched		Non-Prefetched	
	FLASH	Hardwired	FLASH	Hardwired
Memory stall time (%)	0.52		13.14	
Execution time (sys. cycles x10^6)	2.15	2.00	2.38	2.16
Effective CRML (sys. cycles)	117.61	92.37	117.62	92.35
Hardwired speedup (%)	7.55	N/A	10.26	N/A
Latency component (%)	8.7	N/A	100.0	N/A
Occupancy component (%)	91.3	N/A	0.0	N/A

6.3.3 FFT Performance with the Dynamic Pointer Allocation Protocol

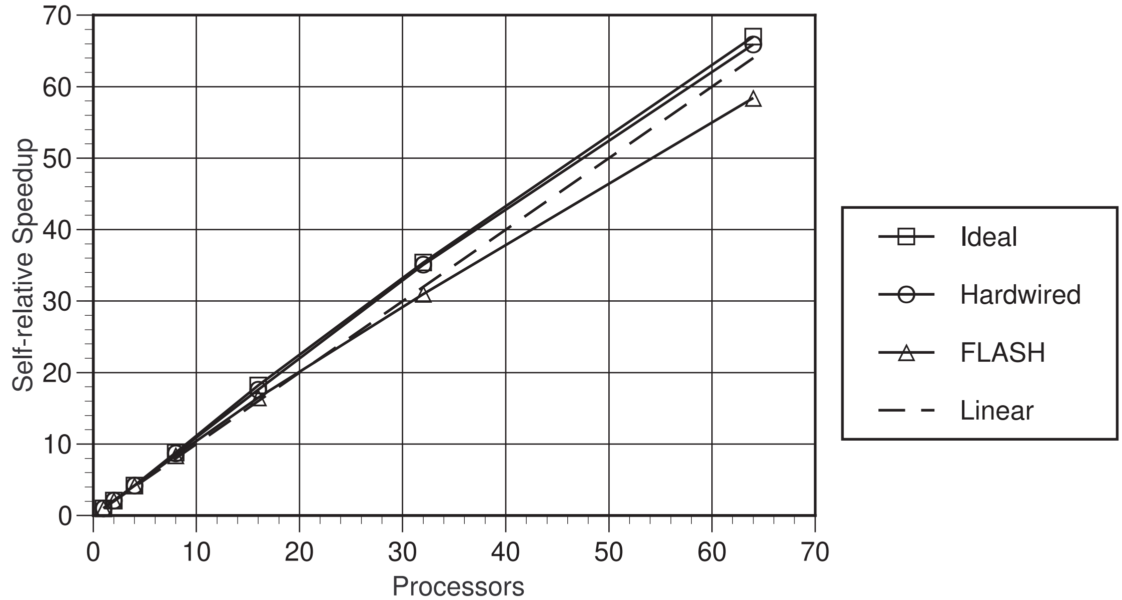
One difference between the Barnes and FFT results is the magnitude of the performance gap between FLASH and the hardwired machine at 64 processors. In the discussion above, this difference was attributed to synchronization overhead and the switch to a coarsevector protocol rather than bitvector. Both machines suffer this increased protocol processing load at 64 processors with the coarsevector protocol but, overall, the FLASH machine handles the additional protocol processing overhead less effectively, giving rise to the increased performance gap.

However, the same flexibility that causes the performance gap can be used to reclaim some of the performance loss, a practical demonstration of one advantage of flexibility. Specifically, if the dynamic pointer allocation protocol is substituted for the coarsevector protocol, FLASH's performance at 64 CPUs, shown in Figure 6.8, is significantly more competitive. At 64 CPUs, the hardwired machine is approximately 15% faster than FLASH versus almost 30% faster with coarsevector; performance at lower CPU counts — 32, for example — is worse, supporting the relative comparison of the bitvector and dynamic pointer allocation protocols from Chapter 5. The superior characteristics of dynamic pointer allocation at larger processor counts, as discussed in Section 5.3, lead to the improved performance.

6.3.4 LU

Like FFT, LU is a kernel. It factors an N-by-N dense matrix into the product of a lower triangular matrix and an upper triangular matrix. The matrix factoring is blocked to make more effective use of the processor caches and to reduce the overall amount of data communicated among the processing nodes. Figure 6.9 shows the speedups for LU and Table 6.12 shows the execution characteristics on FLASH. As with FFT, both prefetched and non-prefetched versions of LU were simulated, but only the prefetched results are included here.

The results for LU are almost identical for all three machines. The low processor cache miss rates, along with the majority of memory references to clean lines — a case for which the protocol is optimized — both help FLASH to maintain competitive performance. In addition, with processor utilizations of 97–99% across all processor counts, very little time is spent stalled on the memory system. This observation is supported by the CRML data, presented in Table 6.13, which show that overall memory system stall time is low and that the small performance difference between FLASH and the hardwired machine is due to latency effects rather than occupancy.



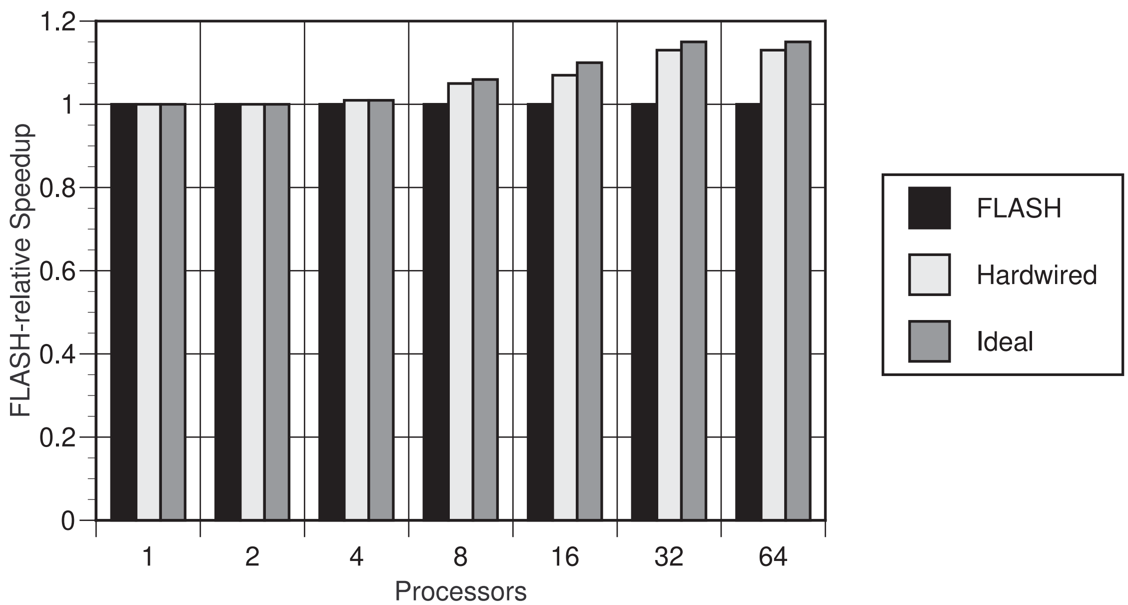
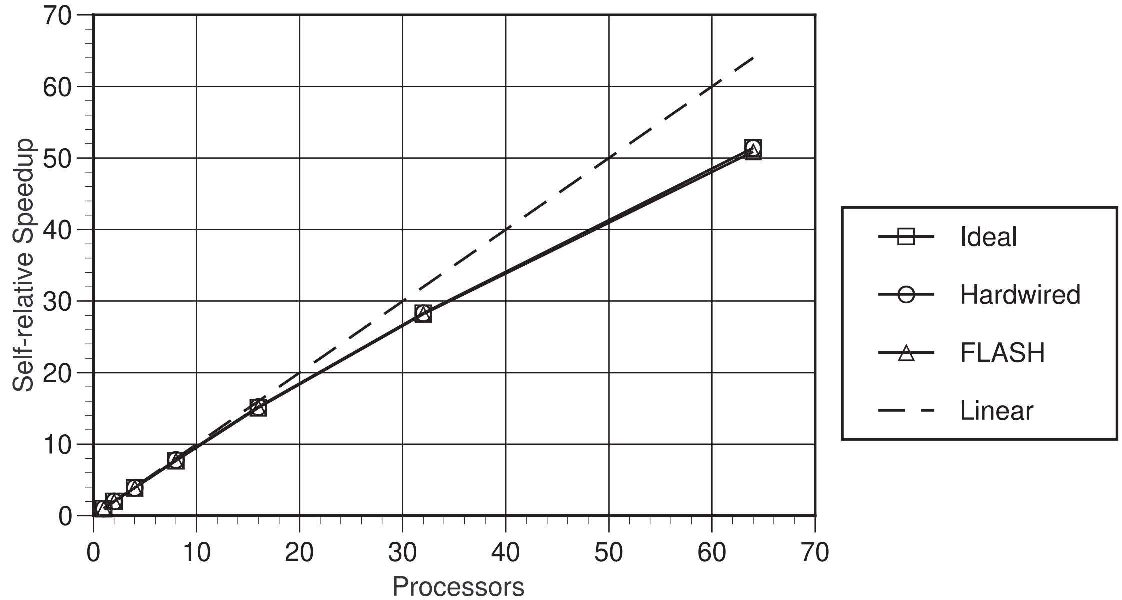


Figure 6.8 FFT speedups with dynamic pointer allocation protocol



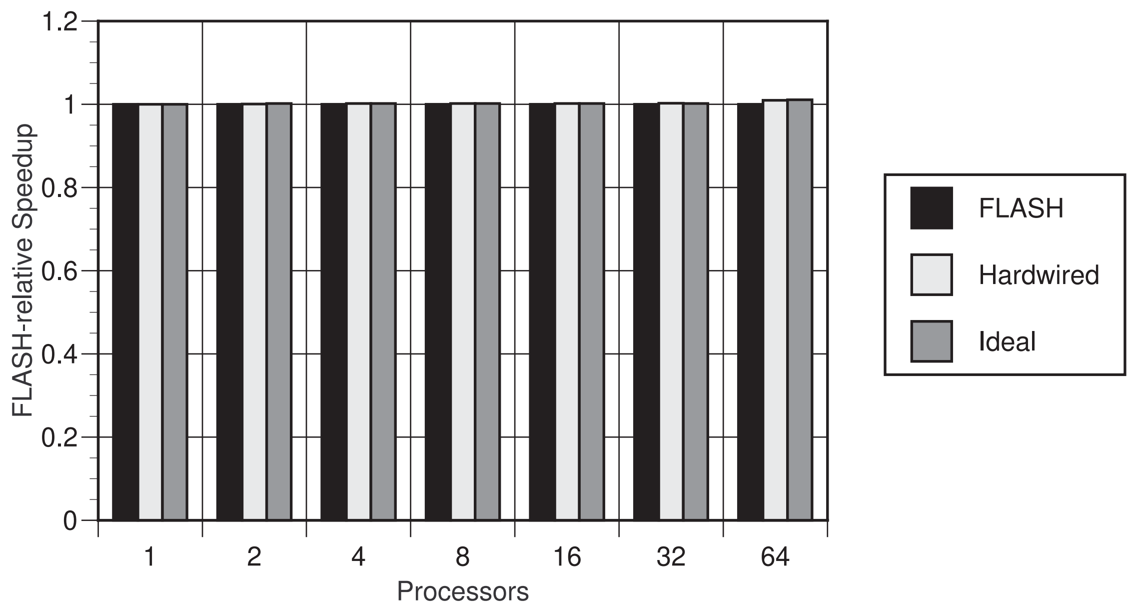


Figure 6.9 LU self-relative (top) and FLASH-relative (bottom) speedups

Table 6.12 LU execution characteristics on FLASH

Execution Characteristic	Processor Count						
	1	**2**	**4**	**8**	**16**	**32**	**64**
Speedup	1.0	2.0	3.9	7.7	15.1	28.2	50.9
Total exec. time (millions)	802.41	407.36	205.35	103.66	53.04	28.42	15.76
Processor util. (%)	97.7	97.8	98.5	99.3	99.6	99.3	98.1
Sync. stall (%)	0.0	0.0	0.0	0.0	0.0	0.1	0.7
Cache reads (%)	67.8	68.1	68.5	69.0	69.5	70.7	72.2
Cache writes (%)	32.1	31.8	31.5	31.0	30.4	29.3	27.8
Cache miss rate (%)	0.06	0.06	0.04	0.02	0.01	0.01	0.02
Rds, lcl, clean (%)	100.0	94.5	88.5	63.1	26.4	16.4	11.8
Rds, lcl, d. rem (%)	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Rds, rem, clean (%)	0.0	2.8	3.6	21.2	50.8	67.8	76.2
Rds, rem, d. home (%)	0.0	2.7	7.9	15.6	22.7	15.7	11.8
Rds, rem, d. rem (%)	0.0	0.0	0.0	0.0	0.1	0.1	0.2
Avg. read miss lat.	26	27	27	31	45	63	123
Avg. write miss lat.	35	35	35	36	38	40	54
Avg./Max. PP util. (%)	2.8/2.8	2.9/3.0	2.3/2.6	1.2/1.3	0.8/0.8	0.9/1.2	1.5/2.8
Dynamic dual-issue eff.	1.32	1.34	1.34	1.38	1.40	1.42	1.38
Special instr. fraction	0.51	0.51	0.51	0.51	0.52	0.54	0.47
MIC miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MDC miss rate (%)	0.10	0.00	0.00	0.00	0.00	0.00	0.00

Table 6.13 LU 32-processor CRML calculation

Characteristic	FLASH	Hardwired
Memory stall time (%)	0.37	
Execution time (sys. cycles x10^6)	28.42	28.34
Effective CRML (sys. cycles)	88.42	79.19
Hardwired speedup (%)	0.29	N/A
Latency component (%)	100.0	N/A
Occupancy component (%)	0.00	N/A

6.3.5 Ocean

Ocean is a complete application that models large-scale ocean movements based on eddy and boundary currents. It partitions the ocean into a series of grids and then applies a red-black Gauss-Seidel multigrid solver [15]. The grid locations are represented using a 4-D array, which improves data locality and processor cache hit rate. Like the other scientific applications, the Ocean code is well tuned for a DSM system. Both prefetched and non-prefetched versions of the code were simulated; the prefetched results are presented here.

Figure 6.10 presents the speedup data for Ocean, and Table 6.14 shows the corresponding execution characteristics on FLASH. Particularly on FLASH, Ocean's performance degrades rapidly at processor counts above 32. All three machines achieve superlinear self-relative speedup (as a result of the increase in the aggregate system processor cache size as more processors are added) through 32 CPUs, but the speedups fall off rapidly thereafter. The execution characteristics for FLASH help explain why FLASH's performance worsens. First, the processor cache miss rate is higher, leading to more memory requests. At 64 CPUs, the average protocol processor occupancy is 20% with a maximum of 33%, an indication of the high protocol processing load and some degree of hot-spotting. The large synchronization stall time also indicates hot-spotting. As the 64-CPU FFT results demonstrated, this extra protocol processing load is handled more effectively by the hardwired machine, and FLASH's performance therefore becomes worse in comparison.

The same effect is observed in the CRML calculation, shown in Table 6.15. Again, even at 32 CPUs, both memory stall and synchronization stall time are fairly high and protocol processing overhead is significant. FLASH's flexibility hurts performance under these conditions and the execution difference between FLASH and the hardwired machine is a result of latency effects. The latency-dominated CRML results indicate that the high percentage of synchronization stall time is not leading to significant hot-spotting, at least at 32 CPUs. This is not unexpected since Ocean uses tree-based barriers, which help to avoid contention on a single barrier lock at the expense of slightly longer barrier join and release latencies.

Even with the tree-based barriers, synchronization stall time is large at 32 CPUs and, especially, at 64 CPUs. One possible means to reduce the synchronization stall time is to implement a synchronization protocol on MAGIC, thereby eliminating much of the overhead of the R10000-based load-linked/store-conditional primitives. This topic is beyond the scope of this analysis, but is investigated in Heinlein [43].

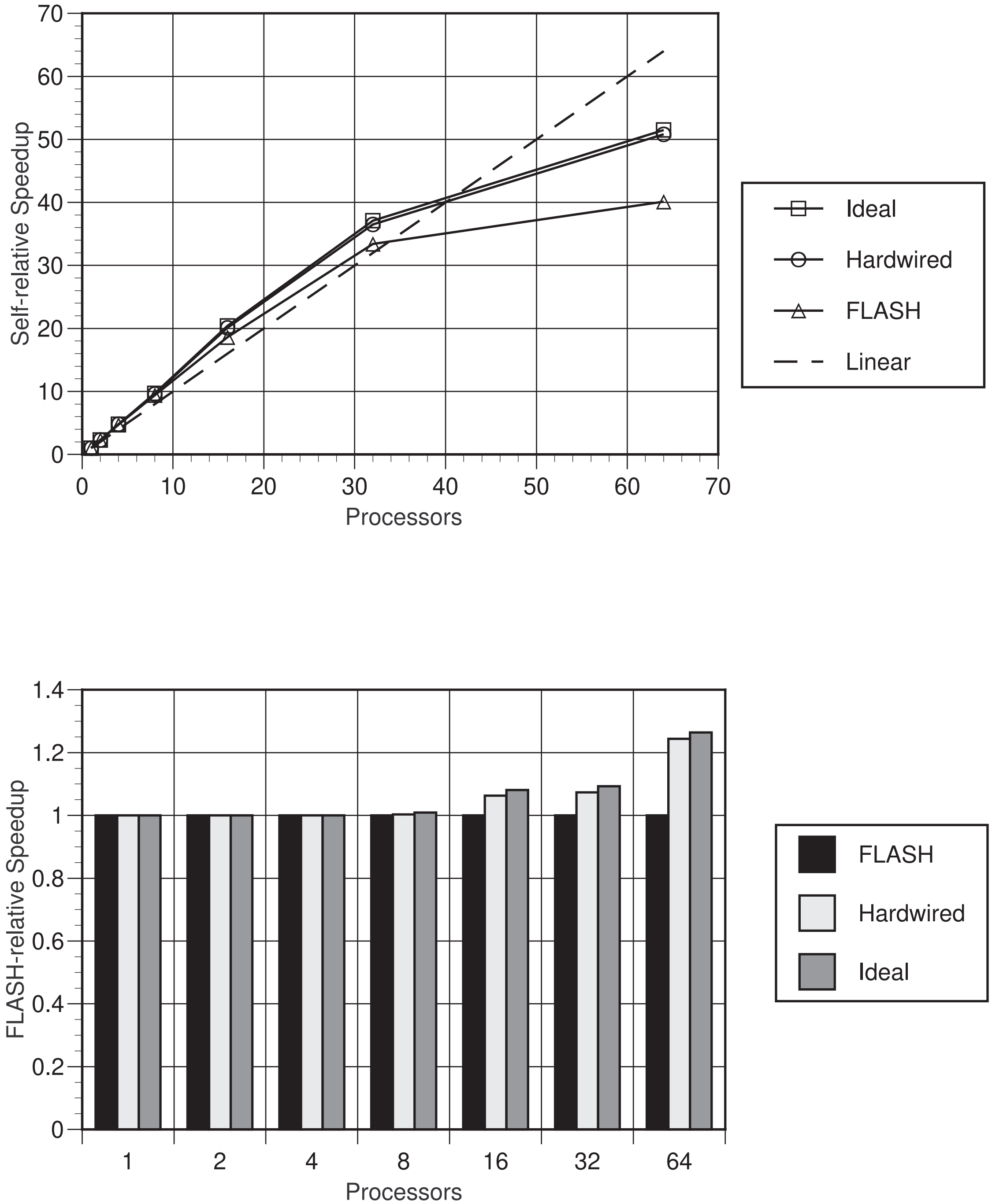


Figure 6.10 Ocean self-relative (top) and FLASH-relative (bottom) speedups

Table 6.14 Ocean execution characteristics on FLASH

Execution Characteristic	Processor Count						
	1	**2**	**4**	**8**	**16**	**32**	**64**
Speedup	1.0	2.3	4.7	9.4	18.6	33.4	40.1
Total exec. time (millions)	201.77	86.56	42.65	21.36	10.84	6.04	5.03
Processor util. (%)	72.4	85.9	87.1	85.2	82.4	72.2	51.8
Sync. stall (%)	0.0	2.5	3.7	9.1	12.6	21.1	41.7
Cache reads (%)	80.0	80.4	80.6	81.4	81.9	83.0	87.2
Cache writes (%)	20.0	19.6	19.4	18.6	18.1	17.0	12.8
Cache miss rate (%)	0.99	0.54	0.46	0.29	0.17	0.12	0.16
Rds, lcl, clean (%)	100.0	98.8	89.4	76.5	31.3	8.2	4.3
Rds, lcl, d. rem (%)	0.0	0.0	0.1	0.4	1.1	2.3	2.6
Rds, rem, clean (%)	0.0	1.0	4.6	9.2	21.1	31.2	27.1
Rds, rem, d. home (%)	0.0	0.2	5.9	13.9	46.5	58.1	66.9
Rds, rem, d. rem (%)	0.0	0.0	0.0	0.1	0.1	0.3	0.3
Avg. read miss lat.	29	27	30	36	72	159	219
Avg. write miss lat.	33	32	33	32	42	43	59
Avg./Max. PP util. (%)	20.7/20.7	16.6/16.6	17.1/17.2	12.8/13.2	12.1/15.7	10.8/14.1	19.9/33.2
Dynamic dual-issue eff.	1.46	1.38	1.36	1.38	1.40	1.40	1.28
Special instr. fraction	0.48	0.49	0.48	0.47	0.46	0.47	0.39
MIC miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MDC miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 6.15 Ocean 32-processor CRML calculation

Characteristic	FLASH	Hardwired
Memory stall time (%)	6.46	
Execution time (sys. cycles x10^6)	6.04	5.62
Effective CRML (sys. cycles)	112.05	88.53
Hardwired speedup (%)	7.34	N/A
Latency component (%)	100.0	N/A
Occupancy component (%)	0.00	N/A

6.3.6 Radix

Radix is the final SPLASH-2 application. It performs an integer radix sort on a contiguous data array using the method described in Blelloch [11], and hence is more properly categorized as a parallel general-purpose, rather than a parallel scientific, application. During each sort iteration, each processor sorts a portion of the array into a local histogram. At the end of the iteration, the local histograms are combined into a global histogram in preparation for the next sorting iteration. The local-to-global histogram conversion is performed using a series of writes — based on the local and global histogram information, each processor writes its local data into the proper place in the global data array. This process leads to communication patterns that are unstructured and bursty, which in turn leads to reduced performance at higher processor counts.

Figure 6.11 shows the speedups for Radix and Table 6.16 shows the execution characteristics on FLASH. Both prefetched and non-prefetched versions of Radix were simulated, but only the prefetched results are included here. As the data demonstrate, speedup for all three machines is poor above 32 CPUs. The algorithm itself and the bursty, unstructured writes both limit performance at larger processor counts, especially since most writes target data that is held dirty in a remote processor's cache. The effect of the communication pattern is visible in the CRML results as well, which are shown in Table 6.17. Bursty, unstructured writes result in a FLASH–hardwired machine performance gap due almost entirely to occupancy effects caused by contention in the memory system.

Since sorting is such a common operation, Radix is an interesting and useful application to evaluate. Ultimately, however, the radix sort algorithm is not well suited for a multiprocessor, regardless of the system's memory latencies. Even the ideal machine barely achieves the desired 0.6 parallel efficiency at 64 processors and, though not shown in the speedup graphs, simulation of a 128-processor run indicates that Radix achieves minimal performance improvement above 64 processors. In this regard Radix is unlike the other scientific applications, which continue to achieve good speedups at 128 CPUs.

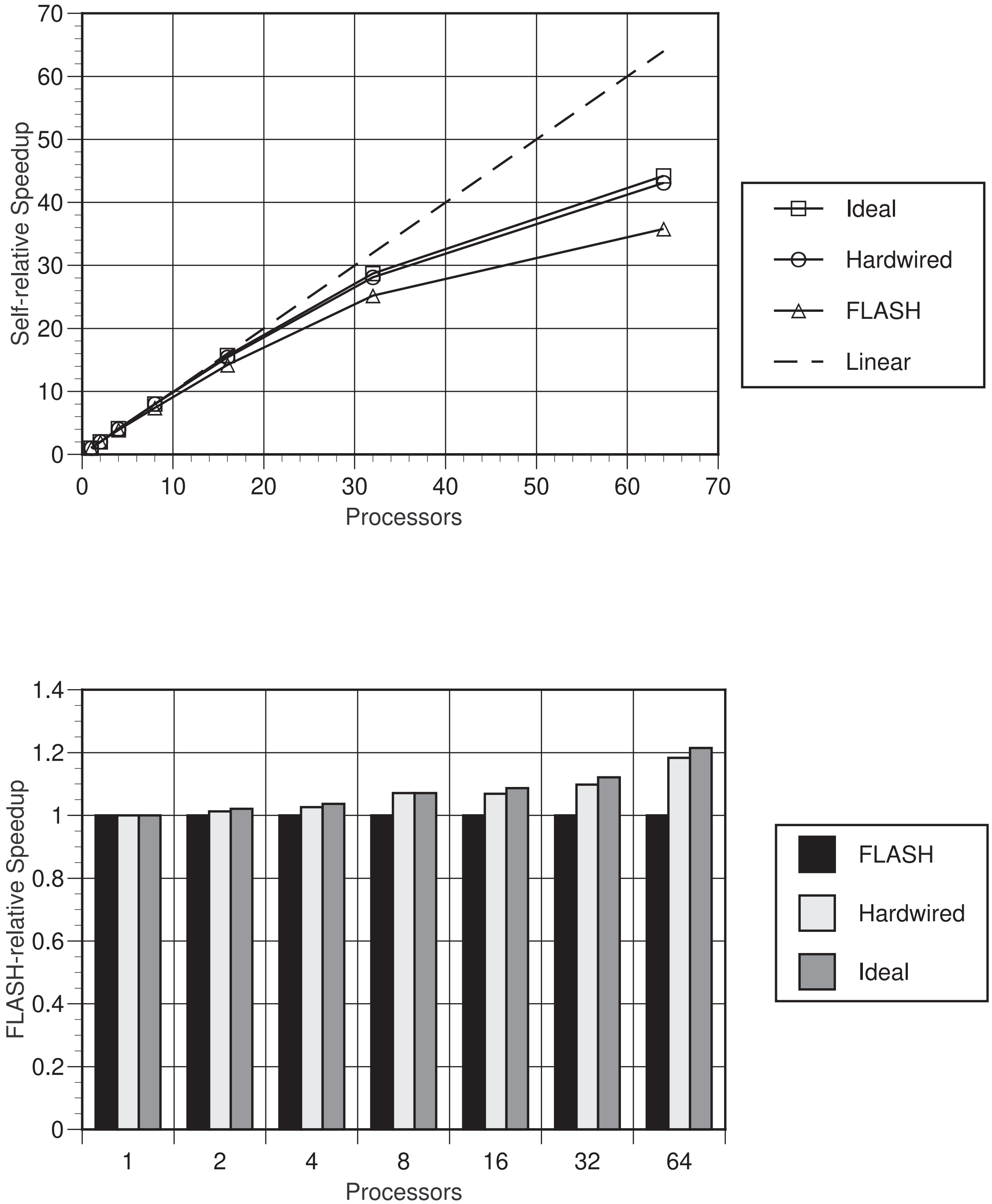


Figure 6.11 Radix self-relative (top) and FLASH-relative (bottom) speedups

Table 6.16 Radix execution characteristics on FLASH

Execution Characteristic	Processor Count						
	1	2	4	8	16	32	64
Speedup	1.0	2.0	3.9	7.4	14.2	25.2	35.8
Total exec. time (millions)	49.27	25.19	12.75	6.65	3.48	1.96	1.38
Processor util. (%)	93.0	92.9	95.9	96.9	96.5	93.4	82.0
Sync. stall (%)	0.0	0.1	0.7	2.1	2.7	5.7	16.3
Cache reads (%)	81.5	81.5	81.5	81.7	81.7	82.1	83.8
Cache writes (%)	18.5	18.4	18.4	18.2	18.2	17.9	16.1
Cache miss rate (%)	0.29	0.28	0.22	0.17	0.17	0.18	0.18
Rds, lcl, clean (%)	100.0	98.8	90.6	53.3	19.6	8.2	3.3
Rds, lcl, d. rem (%)	0.0	0.9	8.2	43.4	74.3	81.1	77.5
Rds, rem, clean (%)	0.0	0.2	1.0	2.3	3.5	5.4	9.7
Rds, rem, d. home (%)	0.0	0.0	0.2	0.8	1.8	3.6	6.5
Rds, rem, d. rem (%)	0.0	0.0	0.0	0.2	0.7	1.8	3.5
Avg. read miss lat.	26	26	28	37	57	103	166
Avg. write miss lat.	30	54	69	88	105	116	141
Avg./Max. PP util. (%)	10.5/10.5	11.8/11.8	12.1/12.4	14.3/15.1	17.5/18.0	18.9/20.2	22.4/27.2
Dynamic dual-issue eff.	1.44	1.44	1.42	1.42	1.42	1.42	1.36
Special instr. fraction	0.48	0.49	0.49	0.49	0.48	0.48	0.43
MIC miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MDC miss rate (%)	0.20	0.20	0.20	0.20	0.20	0.20	0.20

Table 6.17 Radix 32-processor CRML calculation

Characteristic	FLASH	Hardwired
Memory stall time (%)	0.85	
Execution time (sys. cycles x10^6)	1.96	1.78
Effective CRML (sys. cycles)	109.29	92.06
Hardwired speedup (%)	9.77	N/A
Latency component (%)	10.3	N/A
Occupancy component (%)	89.7	N/A

6.4 Multiprogrammed Workload Performance

Whereas the scientific application performance analysis focused on evaluating the three machines across a wide range of processor counts and used speedup as the primary performance metric, the multiprogrammed workloads focus instead on throughput. The primary performance metric still is execution time, but execution time now refers to the total time to complete a set of jobs, each executing on one processor. Self-relative speedup, therefore, is not a sensible metric for the multiprogrammed workloads. Instead, only FLASH-relative speedup is considered. As noted in Section 6.2.2, two multiprogrammed workloads were studied, PMake and Engineering, and the simulation included the operating system as well as the application code.

PMake performs a parallel make of a portion of the gnuchess sources. An 8-way make is used for all processor counts. Engineering simulates the execution of multiple instantiations of FlashLite and the Chronologics VCS Verilog simulator. Unlike PMake, the Engineering workload varies with the processor count. Table 6.18 lists the exact combination of FlashLite and VCS jobs used at each processor count.

Table 6.18 Engineering workload parameters

Processors	Number of FlashLite Jobs	Number of VCS Jobs
1	1	1
2	2	2
4	4	4
8	6	6

The performance of the two multiprogrammed workloads reveals a very different realm of system operation, one in which memory contention and hot-spotting lead to significant performance loss, especially when FLASH is compared to the hardwired machine. Although both the PMake and Engineering workloads were simulated, the results for both workloads are very similar, so only the Engineering results are presented here.

Figure 6.12 shows the FLASH-relative speedups for the Engineering workload and Table 6.19 lists the execution characteristics on FLASH. All Engineering runs use a zero-latency disk model; a realistic disk model masks some important performance characteristics because all three machines spin in the I/O idle loop (waiting for disk operations to complete) at the same speed.

The speedup figure shows that the hardwired machine is 60% faster than FLASH at 8 CPUs. This is a significantly different result from those observed with the scientific

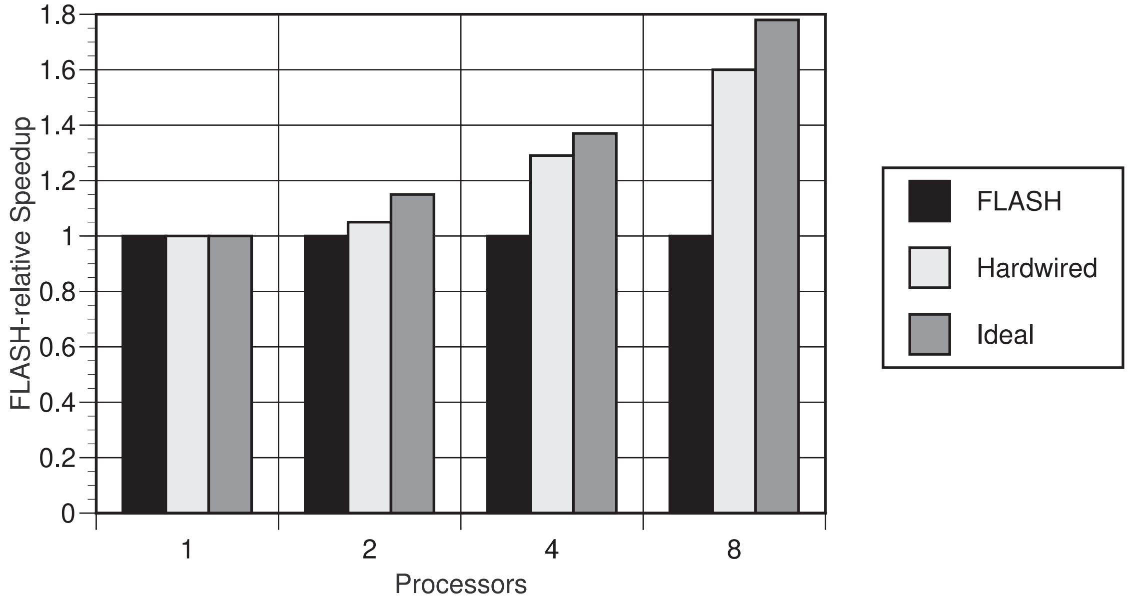


Figure 6.12 Engineering FLASH-relative speedups (zero-latency disk model)

Table 6.19 Engineering execution characteristics on FLASH

Execution Characteristic	Processor Count			
	1	2	4	8
Total exec. time (millions)	70.25	83.46	117.37	137.96
Processor util. (%)	81.2	75.1	61.5	65.9
Cache reads (%)	60.5	61.7	65.9	80.3
Cache writes (%)	39.5	38.3	34.1	19.7
Cache miss rate (%)	0.45	0.46	0.42	0.24
Rds, lcl, clean (%)	100.0	67.0	32.0	20.6
Rds, lcl, d. rem (%)	0.0	0.5	0.3	0.7
Rds, rem, clean (%)	0.0	32.0	66.5	75.6
Rds, rem, d. home (%)	0.0	0.5	0.5	0.7
Rds, rem, d. rem (%)	0.0	0.0	0.7	2.5
Avg. read miss lat.	31	49	103	138
Avg. write miss lat.	34	54	90	117
Avg./Max. PP util. (%)	16.7/16.7	20.7/35.5	26.4/75.1	23.9/75.3
Dynamic dual-issue eff.	1.10	1.20	1.30	1.34
Special instr. fraction	0.51	0.52	0.45	0.41
MIC miss rate (%)	0.00	0.10	0.10	0.10
MDC miss rate (%)	0.60	1.40	0.70	0.60

applications in which the difference between FLASH and the hardwired machine was less than 10% for 16 or fewer processors.

A closer analysis of the memory references reveals that the marked execution time difference is caused by tremendous hot-spotting on a single IRIX kernel lock called *memory_lock*. This lock protects almost the entire virtual memory subsystem, and must be acquired whenever a process is created or destroyed, or when a process allocates memory or otherwise causes a change to its virtual memory mappings. As a result, this lock is highly contended, leading to substantial hot-spotting and protocol processor occupancy at memory_lock's home node. In fact, PP occupancy is over 75% for the 8-CPU run according to Table 6.19, an extremely high value. Because of the lock contention, as well as the OS's use of the load-linked/store-conditional primitives to perform the lock/unlock operations, over 50% of the requests for the lock are negatively acknowledged, further increasing the protocol processing load. And, whenever the lock is released via a store conditional, a slew of invalidate requests must be sent.

Overall, these factors combine to yield a mean memory_lock acquire time on FLASH of about 2800 system cycles versus just over 700 cycles for the hardwired machine. As was the case for the coarsevector 64-CPU FFT performance, the additional protocol processing load caused by memory_lock is managed more effectively by the hardwired machine. On FLASH, the additional processing load causes a substantial decrease in system performance, an indication of how severe hot-spotting greatly exacerbates the effect of FLASH's additional protocol processing overhead. Further investigation showed that the protocol processor would have to run at four times its normal clock rate for FLASH to yield the same performance as the hardwired machine. As noted above, the PMake workload exhibits the same performance trends and the same hot-spotting on memory_lock.

6.5 Summary: Performance Robustness

The performance of FLASH on the Engineering workload highlights an important point about the performance cost of flexibility: although FLASH can achieve performance competitive with the hardwired machine, its performance is not as robust. That is, for applications that are well-tuned for a distributed shared memory system — as all the scientific applications were — FLASH yields competitive performance across a wide range of processor counts, demonstrating the scalability and effectiveness of the FLASH system and MAGIC architecture. These applications exhibit good load balancing, minimal memory hot-spotting, and often specify explicit data placements to further improve performance. With such well-behaved applications, the additional latency and occupancy of MAGIC does not translate into a significant overall application performance difference.

When the application characteristics are not so well-behaved, however, the performance of FLASH — that is, of a system using a flexible node controller — falls off much more quickly than the performance of the hardwired machine. Figure 6.13 illustrates this point by plotting the relative execution times of the three machines for the 8-CPU PMake and 32-CPU FFT applications. The Y axis shows the FLASH-relative execution time while the X axis shows the relative speed of the FLASH protocol processor. Varying the PP speed effectively increases or decreases the amount of protocol processing overhead, thereby modeling how much overhead the flexible node controller adds. A 4x PP, for instance, adds so little protocol processing overhead that its performance is almost identical to the hardwired machine's.

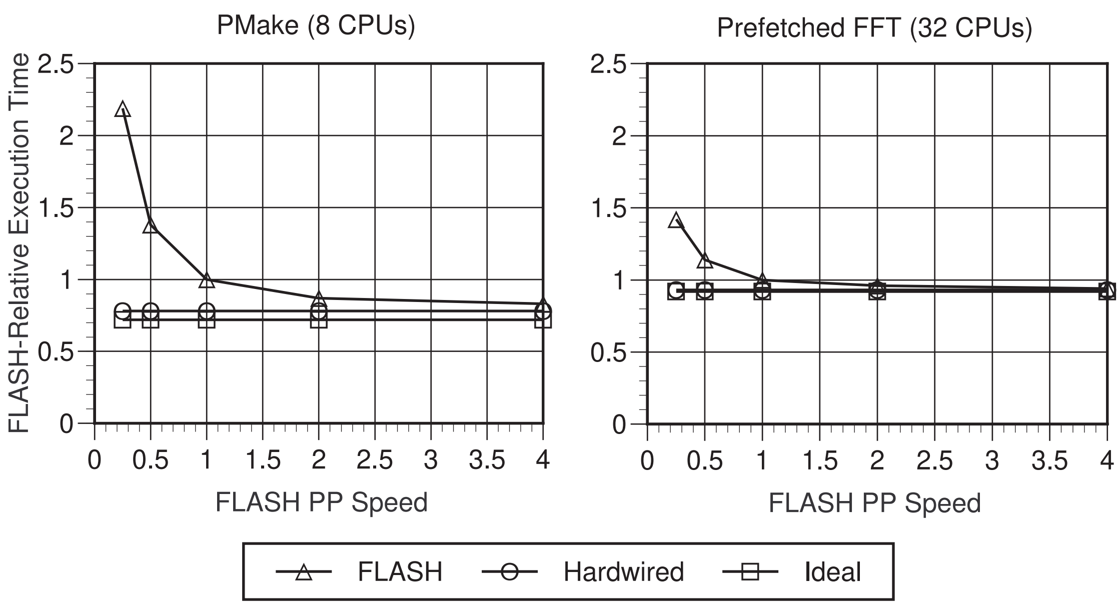


Figure 6.13 FLASH-relative performance versus PP speed

The curves in Figure 6.13 contain both good and bad news for FLASH. The bad news is that FLASH's location at the knee of the curve means that when applications (or the OS) depart from the well-behaved memory reference and load-balancing characteristics exhibited by the scientific applications, FLASH's performance will degrade more suddenly than the hardwired machine's. Clearly, both machines require well-tuned applications to maximize system performance, and the performance of both FLASH and the hardwired machine will be reduced by poor application behavior. However, FLASH's performance will decrease more quickly.

The good news is that the execution time curves have a pronounced knee, and the 1x protocol processor yields performance at the knee of the curve. This indicates that all the effort spent on optimizing the MAGIC architecture for protocol processing and all the additional hardware and architectural features that were added as a result were of significant benefit. In fact, these features are what allows FLASH to achieve competitive performance. In effect, these features moved FLASH to the knee of the performance curve, thereby yielding a system and node controller architecture that can achieve competitive performance yet still employ a flexible approach to protocol processing. This result also indicates that a protocol engine based on an off-the-shelf processor core, without the architectural specialization and hardware support present in MAGIC, would have unacceptably poor performance.

FLASH cannot as a rule achieve protocol processing overheads as low as a machine using a hardwired node controller, and FLASH's higher overheads lead to a lack of robustness in overall system performance. However, FLASH's approach to flexible protocol processing, embodied in the MAGIC architecture, does allows it to achieve competitive performance. Identifying the protocol processing bottlenecks and addressing them with hardware and architectural additions to the protocol processor were crucial in reaching this goal.

Chapter 7
Related Work

A number of multiprocessor systems share with FLASH the goal of delivering scalable performance across a range of applications and processor counts. This chapter compares and contrasts some of these machines to FLASH. With the rising popularity of multiprocessors and the tremendous amount of research activity in the field, a comprehensive discussion of related system designs is beyond the scope of this chapter. Instead, the focus is on distributed systems that share a basic system architecture consisting of a collection of node boards connected via an interconnection network. Both academic and commercial designs are discussed, helping to illustrate the different design tradeoffs appropriate for research and commercial systems.

To stress the importance of the communication paradigm in the design of a multiprocessor, the related systems are categorized as clustered systems, private memory systems, non-coherent shared memory systems, and cache-coherent shared memory systems. A more precise description of each category is found at the start of the corresponding section below.

7.1 Clustered Systems

Clustered systems are those in which the "node boards" are more properly described as workstations. That is, the basic architecture of a clustered system is to use a high-performance network to connect a collection of workstations into a cohesive multiprocessor. Of course, a group of workstations connected by Ethernet is technically a multiprocessor by this definition; the key aspects of a clustered system are the use of a networking technology with significantly higher performance than the ubiquitous Ethernet and the presence of additional software designed to facilitate inter-node communication and the creation and coordination of parallel application processes.

The appeal of clustered systems is their cost effectiveness. By leveraging commercial uniprocessor or small-scale bus-based multiprocessor workstations as processing nodes, a clustered system can realize all the economies of scale associated with the production of the workstations. Often, a similar approach is taken for the interconnection network as well by utilizing a commercially-available local area network technology rather than a network designed specifically for the clustered system. In both cases, not only is the cost of the system minimized, but so too is the design time; off-the-shelf node and interconnection network components mean that the design task consists mainly of system integration and software development.

Along with the cost and design time advantages, however, comes a corresponding performance disadvantage. Because neither the nodes nor, often, the interconnect is designed specifically for distributed multiprocessing, some compromises in overall system performance are inevitable. In addition, the coupling between the processors and the network typically is achieved by attaching a network interface adapter to the system I/O bus, increasing latency and further reducing performance.

7.1.1 IBM SP2

An SP2 system [3, 99] consists of a collection of rack-mounted IBM RS/6000 workstations connected via a custom-developed multistage interconnection network based on the Vulcan switch [102]. A network interface card plugs into each workstation's MicroChannel I/O bus to provide a connection to the network. The Vulcan switch chips support a raw network bandwidth of 40 MB/s, significantly higher than 10 Mbps or even 100 Mbps Ethernet, and comparable to the network bandwidth available in systems such as DASH that were developed at approximately the same time. Hence, although the SP2 nodes are workstations, the system differs from many clustered systems (such as the NOW system discussed below) in that it uses a custom network designed specifically as a multiprocessor interconnect.

The network adapter card contains a number of components to facilitate DMA transfers between the network and a node’s local memory. Among these components are a dedicated memory and switch management ASIC (MSMU) that provides the physical, link, and network layer interface to the network and an Intel i860 processor that performs protocol processing activities such as initiating and controlling DMA transfers. The SP2 supports only the message passing communication paradigm. Block transfers are managed and implemented cooperatively by the compute processors, DMA engines, and i860 protocol processors at both the source and destination nodes.

With a processor in its network adapter, the SP2, like FLASH, provides a programmable means for controlling inter-node communication; in fact, many of the changes between the SP1 network adapter and the SP2 version were to address performance bottlenecks similar to those identified during the development of the MAGIC architecture. Ultimately, however, the location of the SP2 network adapter on the MicroChannel I/O bus limits its performance and leads to increased latency to communicate between the local memory and the network. This is a problem common to most clustered systems since the design of the workstation-based processing node cannot be altered. As a result, the performance of message passing may be acceptable for large block transfers in which the additional latency of initializing and managing the transfer can be amortized across many data bytes, but the performance of finer-grained communication will be significantly lower than on a system, such as FLASH, that more closely couples the network adapter to the compute processor and local memory.

7.1.2 Princeton SHRIMP

Like the SP2, SHRIMP [12, 30] nodes are built from commodity uniprocessors; SHRIMP uses Pentium-based personal computers as the processing nodes. A custom-designed network adapter card connected to the node's system bus provides a connection to the interconnection network. Also like the SP2, the initial SHRIMP system uses a custom-designed network, specifically, the same interconnection network as the Intel Paragon multiprocessor. Since this network is optimized for multiprocessor use, its performance is considerably higher than off-the-shelf LAN solutions.

SHRIMP implements a message passing communication paradigm that strives for low send initiation cost. Low-cost sends are provided by hardware on the network adapter[1] which snoops on all processor memory writes and determines if the write should be propagated to another node. If so, the write data is copied from the bus and forwarded to the proper destination node. The user, via a system call, establishes the mappings between local memory and destination nodes, and this information is stored in the network adapter. Actual sends, therefore, require no special activity in the user's code — the network adapter will snoop on the system bus and transfer data to remote nodes according to the previously-established mappings. A variety of data transport policies allows the user to optimize the message transfers, such as by specifying that data be accumulated in the network adapter before transmission, thereby improving network utilization.

Although the SHRIMP message passing system obtains significant latency reductions to send data, it requires that all writes be visible on the system bus (write-through caching must be used) and supports only limited mappings between local and remote memory. FLASH's message passing implementation uses a similar technique, namely a decoupling of message transfer from the initialization of the local and remote memory mappings, but in a more general framework. By using a programmable engine, FLASH can support a

1. The SHRIMP team has developed two network adapters, SHRIMP-I and SHRIMP-II. The discussion above applies to both designs.

SHRIMP-style message passing functionality while also supporting cache-coherent shared memory and additional forms of bulk data transfer.

7.1.3 Network of Workstations (NOW)

As proposed by a group at the University of California at Berkeley, a NOW system comprises a collection of off-the-shelf workstations connected by an off-the-shelf LAN networking technology such as ATM [6, 25]. By using only commodity components, the cost of a NOW system is expected to be less than custom-designed systems with comparable peak aggregate performance. In addition to its cost advantages, the NOW proposal also is based on the increasing performance of commodity networking components. Just as the rising performance of off-the-shelf microprocessors has led to their dominance in virtually all areas of computing, the NOW proponents argue that a similar transition will occur in networks.

Currently, however, the performance of truly commodity networking technology is too low to serve as effective multiprocessor interconnect. The Berkeley NOW prototype system uses a 160 MB/s Myrinet network developed by Myricom [13] to connect a collection of Sun Ultra 170 workstations. Certainly the Sun workstations are commodity components. But, even though Myricom offers its networking technology commercially, Myrinet is not at present a commodity networking solution. Consequently, the NOW prototype is best viewed as hybrid machine that combines commodity processing nodes with a specialized network.

The NOW prototype supports the message passing communication paradigm and provides both MPI-based [106] bulk data transfer facilities and support for active messages [105]. The active messaging facility achieves user-to-user latency of approximately 10 µs [25], about four times larger than FLASH can achieve for similar operations [42]. Thus, even with the high-speed network, NOW latencies are fairly high and best performance on the NOW system is achieved for applications that perform minimal communication. For example, the system achieves good speedups on LU and on a block tridiagonal (BT) solver, but neither of these applications is communication intensive; LU hardly communicates at all (see Table 6.12), and the BT solver is designed specifically to maximize communication granularity and minimize the number of messages sent.

Overall, the NOW solution has undeniable economic advantages. From a performance perspective, however, the latency and bandwidth of current commodity networks and their lack of close coupling to the compute nodes are insufficient to support fine-grained communication and thus limit the types of applications that will run well on a NOW system.

Many of the parallel applications that performed well on FLASH, for example, are unlikely to achieve comparable performance on a NOW machine.

7.2 Private Memory Systems

Like clustered systems, private memory systems provide a message passing communication paradigm. Although these systems are similar to clustered systems, they are characterized by processing nodes and interconnect technology designed specifically for use in a multiprocessor. The result is closer coupling between nodes and the network and the promise of correspondingly higher performance.

7.2.1 Intel Paragon

The Paragon [53] is perhaps the most commercially successful private memory system. Each Paragon node contains two Intel i860 processors[2] along with a network interface chip and local memory, all connected to a common shared bus. One processor acts as the computation processor and the other acts as the communication controller. Messages arriving at the node cause a handler to be dispatched on the communication processor, which processes the message header and arranges for any necessary DMA of message data into the local memory. Message sends by the compute processor also invoke a handler on the communication processor, which composes the message header, forwards it to the network interface chip, and arranges for any message data to be transferred.

With separate communication and computation processors, the Paragon node naturally allows overlap of communication and computation, a key advantage of message passing based communication. Several message passing protocols are provided, including the widely-used MPI protocol [106]. No form of shared memory is implemented — all communication among the processing nodes is requested explicitly.

The Paragon design is similar to FLASH in that it uses a programmable engine for protocol processing. However, the use of a shared bus means that the coupling between the various node components is not particularly close, and the i860 processor is not specialized to protocol processing. As a result, end-to-end message delivery latencies are close to 10 µs. Some of this latency can be ascribed to inefficiencies in the message passing library software; even so, the Paragon's performance is likely to be poor for codes requiring fine-grained communication, and a Paragon system is unlikely to be capable of providing any reasonable support for shared memory, two goals the FLASH system is designed to

2. Later designs use Intel Pentium Pro processors.

address. These were not among the goals the Paragon system was designed to address, however. Instead, the design is optimized for codes characterized by coarse-grained communication of relatively large amounts of data. For this type of communication, the Paragon design will work well. The large amount of data accompanying a typical message allows the protocol processing overhead to be amortized effectively, helping to alleviate the performance loss caused by the use of a general-purpose CPU for protocol processing and the lack of close coupling among the compute processor, communication processor, and network.

7.2.2 MIT J-Machine

Unlike the Paragon, which provides separate compute and communication processors and routers, the J-Machine [80, 100] integrates computation, communication, and routing functionality into a single chip, called the Message-Driven Processor (MDP). The MDP contains a processor core along with memory-management, routing, network interface, and DRAM controller modules to yield almost a complete multiprocessor node on a single chip. All that is necessary to complete the node, in addition to the MDP itself, is local DRAM.

By integrating functions that in most multiprocessor designs are provided by several chips, the MDP provides extremely tight coupling between the compute processor and the network. In essence, the MDP takes the need to provide close coupling among the node components to an extreme — not only are the compute and communication processors one and the same, but the network interface is integrated almost directly into the MDP processor core's datapath. Messages can be sent directly into the network from the MDP memory or register file, and messages received from the network cause a thread to be created and dispatched at the destination node, the message data to be transferred into the local node's memory, and a pointer to the message data to be stored in a processor register.

Such close coupling recognizes that the essence of multiprocessing is communication — the more efficiently multiprocessor nodes can communicate data and synchronization information, the better overall system performance will be. J-Machine nodes communicate using message passing primitives, although the MDP memory management hardware supports global shared memory as well. Because of the MDP integration, message latencies are reduced considerably versus comparable systems such as the Thinking Machines CM-5.

The MDP design of using the same processor to perform both communication (protocol) processing as well as computation is interesting, and leads naturally to the tight coupling among the node board components described above. From a performance perspective, however, the benefits are less clear. First, protocol processing represents a significant processing load in itself; requiring the same processor to perform both protocol processing and computation necessarily slows computational performance and makes overlapping communication and computation more difficult. Second, best computational performance is achieved only by using commercial microprocessors. The MDP architecture precludes the use of off-the-shelf processors since such processors do not permit the network and protocol processing features to be integrated on the same chip.

Nevertheless, the J-Machine and MDP goals of close coupling between the processor and network are extremely desirable and demonstrate the utility and message latency reduction that can be achieved. One recently announced commercial microprocessor — the Sun UltraSPARC III — already integrates a memory controller. Future microprocessors surely will integrate network functionality as well, perhaps permitting a J-Machine style system to be constructed from commercial microprocessor technology. A follow-on to the J-Machine, the M-Machine [32], has similar architectural goals, though with a considerably more sophisticated processor core.

7.3 Non-Cache-Coherent Shared Memory Systems

Shared memory systems, as the name implies, implement a shared memory communication paradigm. These systems typically are designed as multiprocessors from the start, with processing nodes and interconnection networks designed specifically for operation as a multiprocessor. Inter-processor communication latency usually is shorter than in the private memory and clustered systems discussed above, an important component of supporting shared memory efficiently. Some shared memory systems also support cache coherence. This section covers machines that do not provide cache coherence; Section 7.4 presents cache-coherent systems.

7.3.1 Cray T3D and T3E

Both the T3D and T3E systems [7, 94] are shared memory machines targeted at the large-scale scientific application market. Almost all Cray systems have been available in multiprocessor configurations, but the T3D and its successor the T3E mark the first use of a large-scale distributed NUMA architecture based on commodity microprocessors. The T3D and T3E both use DEC Alpha processors, the 21064 in the T3D and the 21164 in the T3E. To perform well in the vector supercomputer arena, these systems must provide support for a large amount of physical memory and, equally important, large bandwidth to memory.

On the T3D, the limitations of the 21064 made achieving these goals a challenge. First, the 21064 supports only 34 bits of physical address space. Second, it supports only one outstanding cache miss. Third, the T3D node contains no second-level cache; the only cache is the 8 KB first-level cache on the 21064 die. These limitations led Cray to provide a "shell" of support hardware around the 21064 to extend its addressing and bandwidth capabilities. The shell comprises several components, including the DTB Annex, which maps 34-bit physical addresses from the processor into a 64-bit global address, effectively extending the physical address space. The shell also provides a set of global fetch-and-increment registers, a prefetch FIFO for supporting prefetching and pipelining of remote memory fetches, a DMA-based block transfer engine, and a dedicated barrier network. The combination of the shell features yields a rich set of mechanisms for supporting a very large physical memory as well as extensive prefetching and block transfer support to maximize remote memory fetch bandwidth. A custom-designed 3-D torus network connects the dual-processor nodes.

Unfortunately, actual use of the T3D showed that the shell provided too much functionality. The remote memory prefetch and fast network both were used extensively. The design of the DTB Annex, however, proved too inflexible to be exploited efficiently by the compiler, and the dedicated barrier network, while used, did not yield significant application performance improvement and incurred a high implementation cost. The high startup cost for the block transfer engine limited its use as well. From these observations, the T3E design emerged. The T3E uses a faster processor, the 21164, and continues to surround it with a shell of support hardware. The T3E shell is simpler yet more flexible. It replaced the DTB Annex and block transfer engine with the more flexible E-registers and centrifuge to support remote memory access pipelining and scatter/gather. The barrier network is eliminated entirely. Like the T3D, the T3E contains no board-level cache; the only caches are the 8 KB first-level and 96 KB second-level caches on the 21164.

Neither the T3D nor T3E provides cache-coherent shared memory; an underlying design assumption is that the scientific applications meant to run well on the machines can do so without cache coherence or by managing coherence at a user level within the application. While FLASH shares the goal of providing a shared memory environment, FLASH focuses on hardware-based cache-coherent shared memory. The ease of programming in a cache-coherent system, especially the ability to incrementally tune applications developed for a bus-based multiprocessor, argues strongly for supporting cache coherence, especially if effective support for general-purpose computation is desired. FLASH also seeks to provide the functionality of the T3D and T3E support shell using the programmable engine in MAGIC rather than dedicated, fixed-function components.

7.3.2 Tera

The Tera system [4, 5] contains processing nodes, I/O nodes, and memory nodes connected via a high-performance multistage interconnection network. Unlike all other systems discussed in this chapter, the Tera machine does not use off-the-shelf microprocessors. Instead, it uses a custom-developed multithreaded processor. Multithreading allows a single processor to execute many different threads of control and helps to hide memory latency by switching from a thread waiting for data to return from memory to a thread that is ready for execution. In this way, memory access latency is essentially an extension of the CPU pipeline and if a sufficient number of threads is available, memory latency can be hidden entirely with no loss of aggregate performance.

Unfortunately, the Tera system needs many threads to hide memory latency effectively. With an estimated memory latency of 70 processor cycles and no processor caches, each CPU must be assigned at least 70 threads to allow memory latency to be hidden. A feature called explicit dependence lookahead helps reduce the number of threads required by allowing independent instructions (including loads) from the same thread to be executed before previous instructions from the same thread complete. Even with an average of four independent instructions between dependent ones, however, about 20 threads per CPU are needed. In a 64-processor system, then, 1280 threads are needed. While this degree of parallelism is present in some applications, depending on so much parallelism is risky, especially since system performance will degrade rapidly as the degree of parallelism decreases.

Comparing FLASH to Tera is difficult beyond the common support for shared memory. Rather than relying on multithreading and a large amount of parallelism, FLASH relies on the processor caches and memory latency hiding techniques such as prefetching, non-blocking loads, and multiple outstanding cache misses to maintain good processor utilization in the face of high remote memory access latencies. Tera's decision not to use an off-the-shelf processor is also questionable. Successful operation of a Tera system was announced late in 1997 [103], yet its processor performance is at best comparable to contemporary microprocessors. Given the microprocessor performance growth rate of about 50% per year, the current Tera processor's performance soon will lag microprocessor performance by a substantial margin.

7.4 Cache-Coherent Shared Memory Systems

This section analyzes systems in that provide a cache-coherent shared memory communication paradigm. Although all these systems are cache-coherent, they employ varying system architectures and coherence protocols. In all cases, however, the systems are designed as multiprocessors from the start, and use node designs and interconnects that are designed specifically for use in a distributed multiprocessor.

7.4.1 SGI Origin 2000

The Origin system [64] is closest to FLASH in architecture; both systems use the same R10000 compute processor and the same interconnection network. Each also uses a single node controller chip to provide connectivity to local memory, the interconnection network, and to a local I/O subsystem. In the Origin, the node controller is completely hardwired and implements a variation of the bitvector/coarsevector protocol described in Section 5.2. Because FLASH and the Origin 2000 are architecturally similar, they serve as a useful comparison for evaluating the tradeoffs of using a programmable protocol engine; Chapter 6 explored this issue in detail. Clearly the FLASH approach yields significantly higher memory access latencies, although the time to fetch a local, clean line is the same. On the other hand, the Origin node controller cannot change to a different protocol, such as dynamic pointer allocation, that may be better suited for larger systems. For a commercial system, the hardwired design probably is more appropriate; Chapter 8 looks at some possibilities for combining the lower latencies of a hardwired design with the flexibility benefits of a programmable engine.

7.4.2 Convex Exemplar and Data General NUMALiiNE

Both of these systems use the Scalable Coherent Interface (SCI) protocol [39, 91] to connect multiprocessor nodes (clusters) into a cache-coherent multiprocessor. Coherence within a cluster is maintained using a non-SCI protocol; SCI is used only for inter-cluster coherence. The Exemplar node, called a hypernode, contains up to 16 PA-8000 processors and eight memory banks, all connected via a crossbar switch [8, 16]. The NUMALiiNE node is smaller, containing up to four Pentium Pro processors and memory, connected via a shared bus [21].

As noted above, the unifying feature of these systems is their use of SCI to maintain system-wide cache coherence. The SCI protocol, like the dynamic pointer allocation protocol presented in Chapter 5, maintains sharing information using a linked list data structure. In the SCI protocol, however, the sharing list is physically distributed among the processing nodes — each line in every CPU's cache contains not only the usual tag and state information, but a linked list sharing entry as well. When a cache receives data in response to a processor read or write request, the linked list entry from the associated cache line is joined into a distributed linked list of sharers maintained by all nodes caching the memory location. As with dynamic pointer allocation, each memory location has a home node where the directory entry is located. For SCI, however, the directory entry contains just a few bits of state and a pointer to the first cache on the sharing list.

The SCI distributed sharer lists offer three major advantages. First, as noted for the dynamic pointer allocation protocol, use of a linked list allows precise sharing information to be maintained even for very large processor counts. Second, by physically storing the sharing information along with the cache line being shared, SCI achieves very low directory memory overhead. Third, by distributing the sharing list, SCI distributes protocol processing as well. The overall protocol processing activity required to service a cache miss is not handled almost entirely by the home node; instead, each node on the sharing list bears some of the protocol processing overhead, helping to reduce hot-spotting.

Unfortunately, the distributed sharer lists also greatly complicate the protocol. With a centralized sharing list, as used by dynamic pointer allocation for example, traversing and updating the sharing list is an atomic operation — no other requests to the same cache line will be serviced until processing of the current request completes. With SCI, this atomicity is impossible since inter-node messages must be sent to cause the sharing list to be manipulated. As a result, SCI incurs substantially greater overall protocol processing overhead — a result of a much more complicated protocol — even though it does help spread the processing load across the system.

An implementation of SCI has been developed for FLASH. Preliminary performance results indicate that the complexity of the SCI protocol causes a significant performance loss at small processor counts (typically fewer than 64). At larger processor counts, SCI is more competitive with dynamic pointer allocation, though overall system performance with SCI still is lower, confirming that SCI's utility is limited to larger system sizes [45]. For very large processor counts, the tradeoffs between SCI and dynamic pointer allocation are not clear and the superior protocol may be as much a function of the application characteristics as the system size.

7.4.3 Wisconsin Typhoon

Typhoon refers more to a class of multiprocessor architectures than to a specific implementation. The Wisconsin researchers have proposed a separation of the interface to a shared memory system and the hardware mechanisms used to implement the interface [82, 93]. Tempest is the name of the interface specification and Typhoon refers to a proposed hardware implementation platform.

Several Typhoon designs have been proposed. The original design [82] proposed the use of a previous-generation SPARC processor core as a protocol processor with various support hardware to provide protection, memory translation, and a network interface. The Typhoon-0 proposal [84] separates the node controller functionality into three separate chips: an off-the-shelf protocol processor, an access control device, and a network interface chip. All three components are connected to the memory bus of a SPARC workstation. The Typhoon-1 proposal [85] is similar to Typhoon-0 but combines the access control device and network interface on one chip. Typhoon and Typhoon-1 are just paper designs and were never built. A Typhoon-0 prototype was constructed using a dual-processor SPARC workstation (one CPU served as the compute processor and the other as the protocol processor), a Myricom network interface chip, and an FPGA-based access control device.

The difficulty with all three Typhoon approaches is their use of off-the-shelf protocol processors. As the results in Chapter 6 demonstrate, the key to using a programmable protocol engine yet achieving performance competitive with a hardwired design is to specialize the protocol engine architecture to the characteristics of protocol code, provide additional hardware support to alleviate performance bottlenecks such as dispatch, and to closely couple the protocol engine to the other node board components. Although the three Typhoon designs achieve these goals to a varying extent, none of the proposals achieves the same comprehensive, integrated solution as the MAGIC chip. Consequently, the performance of the Typhoon systems is likely to be significantly worse than a FLASH system, and is unlikely to be competitive with a hardwired system on any but the least communication-intensive applications. The latency to service a remote miss, for example, is over 300 cycles on the Typhoon-1 design and over 600 cycles on Typhoon-0 [85] (converted into 100 MHz cycles and excluding network transit time). These latencies are approximately 5 and 10 times longer, respectively, than on FLASH (see Figure 6.2). Typhoon protocol processor occupancies also are large, an additional indication that overall system performance is likely to suffer.

The Tempest and Typhoon proposals are a useful way to view the functionality of a multiprocessor node controller, especially one based on a programmable protocol processor. From a performance perspective, however, the proposed use of an off-the-shelf protocol processor and the lack of close coupling among the node components are liable to yield inferior overall system performance.

7.4.4 MIT Alewife

Like FLASH, Alewife [2] aims to support both cache-coherent shared memory and message passing. Each Alewife node contains a modified SPARC compute processor, local memory, and a node controller called the Communication and Memory Management (CMMU) chip. The CMMU basically is a hardwired protocol engine, and implements both the LimitLESS cache coherence protocol [1] and the message passing functionality. An interesting feature of the CMMU design, however, is that it implements only part of the cache coherence protocol. When the CMMU encounters a protocol case it is unable to service — typically the corner cases of the protocol — it traps to the node's compute processor. The compute processor services the protocol operation and then returns control to the CMMU. This feature eases the CMMU logic design because the CMMU needs to implement only the common-case protocol operations.

One difficulty with this approach is that the compute processor must be able to take a trap at any point in its operation, even when it is stalled on a cache miss. This functionality was not present in off-the-shelf SPARC processors, and hence the Alewife team had to modify the SPARC design to provide the necessary trap servicing features and to reduce trap service overhead to only a few cycles to reduce protocol processing overhead. The modified SPARC design is called Sparcle. In addition to the trap logic, Sparcle also supports multiple contexts, which help to tolerate remote memory fetch latency by switching to another context while a remote miss is being serviced.

The Alewife architecture demonstrates one way in which hardwired and programmable protocol processing can be combined. Unfortunately, the Alewife design required modifying the compute processor, an option not available for the FLASH design and, in any event, an option that is increasingly difficult to undertake with contemporary high-speed, full-custom VLSI microprocessor designs. Good performance on Alewife requires that very few protocol operations trap to the local Sparcle processor, since each trap not only incurs additional overhead to service the protocol operation but also interrupts normal computation. Performing protocol processing on the compute processor also greatly increases the occupancy of the protocol operation.

7.4.5 Sequent NUMA-Q

The NUMA-Q system [68, 69] is a good example of how the basic FLASH approach can be applied in a commercial system. Each NUMA-Q node — called a Quad — contains four Intel Pentium Pro processors, an Intel memory controller, local memory, and an I/O controller. All these components are connected to a shared, snoopy bus that maintains cache coherence within the Quad. Also attached to the snoopy bus is a node controller that provides system-wide cache coherence using the SCI protocol described in Section 7.4.2.

The node controller architecture is of particular interest because it is based on a programmable protocol engine similar to the MAGIC design. To optimize the node controller, called the SCI Cache Link Interface Controller (SCLIC) chip, for a commercial setting, its design was specialized to the SCI protocol. In contrast, the MAGIC architecture is designed for generality and maximum flexibility so as to support many protocols efficiently. For a commercial product, however, such flexibility is unnecessary and a faster protocol engine can be designed if only a single protocol needs to be supported.

The SCLIC chip contains a programmable engine based on a simple, three-stage pipeline with a VLIW-style instruction set optimized for the SCI protocol and including bitfield extensions. Unlike MAGIC, the SCLIC chip does not provide an instruction cache; all protocol code is stored on-chip in a dedicated memory large enough to store all protocol code. A data cache is present, although it is used only as a cache for SCI directory entries rather than as a general data cache. In addition, the SCLIC programmable engine supports multiple contexts, and a context is allocated and left pending until the associated memory operation, such as a remote memory read, completes.

MAGIC and the SCLIC chip share a number of major architectural features, most significantly the use of a programmable protocol engine. The SCLIC design illustrates one way in which MAGIC's general protocol processing support can be tailored to a specific protocol, thereby yielding a faster and perhaps smaller design that has performance characteristics suitable for use in a commercial system.

Like the NUMA-Q, the S3.mp system [79], developed at Sun Microsystems, also used a programmable protocol engine. Although the S3.mp was a research machine, it also adopted the approach of specializing the protocol processor to a particular protocol. On the S3.mp, the programmability was exploited by developing a COMA protocol for the system in addition to the original protocol for which the node controller was designed [90].

7.5 Software-Based Shared Memory Systems

Another possibility for providing coherent shared memory is to forgo direct hardware support and instead provide the coherence mechanisms via a combination of operating system and user-level software. Potential advantages of software-based shared memory include the ability to use off-the-shelf workstations and networks, with the accompanying cost and time-to-market benefits, and the ability to more easily tailor the coherence code to the application or system characteristics.

A number of software-based systems have been proposed, including Treadmarks [60], Munin [17], CRL [56], MGS [115], Shasta [92], and SoftFLASH [31]. Generally, these systems operate on a page level and rely on the operating system to detect when coherence operations are required by marking pages read-only or invalid in the TLB. On TLB faults, the proper protocol code is executed to maintain coherence.

The most significant drawback of software-based coherence is the resulting application performance. Although these systems can yield acceptable performance for applications with infrequent, coarse-grained communication and minimal false sharing, the performance on a software-based coherent system for most applications is considerably worse than on any machine using hardware-based coherence [31]. Performance is degraded by the additional processing overhead of the software coherence handlers, which tend to be much more complicated than the handlers (or logic) used in a hardware-based machine. In addition, maintaining coherence at the granularity of a page rather than a cache line is too coarse-grained for many applications and leads to false sharing, increased network traffic, and often requires significant amounts of additional memory to track modifications to read/write pages. On balance, then, software-based coherent shared memory may be appropriate to allow off-the-shelf machines, perhaps even a non-homogeneous collection of workstations, to be networked into a shared memory multiprocessor. For all but the least communication intensive applications, however, the performance is likely to be far worse than on any machine with hardware-based coherence [23, 52].

7.6 Summary

The number of related systems presented in this chapter illustrates both the gaining acceptance of multiprocessors in the computer market as well as the diversity in distributed multiprocessor system architectures. Although several message passing systems are discussed above, pure message passing systems generally have fallen out of favor in the last several years. These systems have achieved impressive scalability, but the difficulties of programming for the message passing paradigm as well as the inability to run existing applications without modification have limited message passing acceptance, particularly in the commercial market. Instead, shared memory systems, particularly cache coherent systems, have become the dominant type of commercial system.

Even within the shared memory design space, however, numerous architectures are possible. Issues such as the design of the individual processing nodes and the choice of protocol clearly are not settled, as demonstrated by the variety and contrast of the approaches described above. This lack of agreement on the system architecture and, particularly, the protocol argues well for a system such as FLASH. The flexibility of the MAGIC chip allows investigation into protocol design tradeoffs and whether both message passing and cache coherence can be used together to improve overall system performance. As system sizes grow, tradeoffs such as these will become ever more important to the efficient use of the huge aggregate computational power of large systems.

Chapter 8

Conclusions and Future Work

As multiprocessors scale to ever larger processor counts, the design of the memory system — the node controller, local memory, and interconnect — becomes increasingly important. Achieving good overall system performance requires a balanced system design that recognizes the need to combine fast individual compute processors with a memory system that allows the machine's raw aggregate processor performance to be exploited effectively. For small-scale systems, attaching all processors and memory modules to a shared, snoopy bus yields good performance and allows cache-coherent shared memory to be provided. As the system size grows, however, and a distributed architecture becomes necessary to overcome the scalability limitations of the shared bus, the memory system design task becomes more difficult.

A distributed memory architecture allows very large systems to be built, but greatly complicates the means by which the processing nodes communicate. Thus the major tasks in designing a large-scale distributed multiprocessor are deciding what communication paradigms the system should provide and how the paradigms should be implemented. The FLASH design focuses on providing the means to evaluate these tradeoffs quantitatively. To permit detailed investigation of communication paradigm and protocol tradeoffs, the FLASH system allows many different protocols to be implemented and tested on an actual machine. The key to achieving this goal is the MAGIC chip, FLASH's node controller, which is based on a flexible, programmable protocol engine, a feature critical to supporting multiple protocols. In addition to supporting multiple protocols, FLASH also is designed to be scalable and to reduce the logic and memory overhead of providing cache-coherent shared memory. At the same time, so that the results from investigations performed on FLASH are applicable to hardwired designs, overall FLASH system performance must be competitive with a system employing a hardwired node controller.

Achieving these goals required a detailed investigation of how to design a programmable protocol engine. To minimize system cost and design time, one reasonable approach is to use an off-the-shelf processor. However, analysis of typical protocol operations showed that this solution yields inferior protocol processing performance. Ultimately, the analysis demonstrated the need for an integrated node controller based on a programmable protocol processor with an instruction set and architecture optimized for the characteristics of protocol code. Along with the protocol processor, the node controller needed to include additional hardware support to reduce data movement and message dispatch overhead. Integrating all this functionality on a single chip closely couples the node controller to the other processing node components, further reducing protocol processing overhead.

Chapter 4 described the resulting MAGIC design and Chapter 6 investigated the performance of the MAGIC chip and the FLASH system on both large-scale parallel applications and multiprogrammed workloads. The performance results indicate that FLASH is scalable — it achieves good speedups across a range of processor counts — and can achieve overall system performance competitive with a machine retaining the same system architecture but using a fully-hardwired node controller. However, FLASH's performance is more sensitive to undesirable application characteristics such as poor load balancing and memory hot-spotting. When an application exhibits these behaviors, the performance of the FLASH system degrades more rapidly than the hardwired machine's, an indication that FLASH's performance, while competitive, is less robust.

Given the FLASH and MAGIC design and the system performance analysis presented in this dissertation, two related areas of future work seem viable: adapting the programmable node controller design to yield more robust system performance and using the FLASH system to perform large-scale system architecture and protocol investigations.

8.1 Node Controller Architecture Alternatives

The MAGIC architecture supports a variety of protocols. As such, the architecture is intentionally very general, optimized only for protocol processing as a whole rather than for a particular protocol or set of protocols. Although the resulting design does yield efficient support for a range of protocols, its lack of specialization leads to higher memory operation latencies and protocol processor occupancies than a design tailored for a particular protocol. By compromising some generality, several alternate node controller architectures could help reduce the latency and occupancy overheads — and thereby improve overall performance and performance robustness — while retaining many of the advantages of a programmable protocol processor core.

One option is to use multiple protocol processors. This option was considered for the MAGIC design but discarded because of implementation constraints. With the rapid improvement in ASIC processes and usable gate count, however, providing two protocol processors on the same die is feasible. The Sequent NUMA-Q designers also considered incorporating two protocol processors [69], and the Sun S3.mp node controller implements two protocol engines, albeit a simpler design than the MAGIC PP.

The key issue in employing multiple protocol processors is how to assign work. With two PPs, for instance, a natural choice is to inspect incoming messages and dispatch those pertaining to remote addresses to one PP and local addresses to the other PP. This division is desirable because, for many cache coherence protocols, only requests to local addresses require access to the directory, meaning that only one PP will be reading and updating the directory. For large parallel jobs with good load balancing, this partition is likely to work well. If the system were to run a collection of uniprocessor jobs, however, and memory were allocated on the same node as the job — in general, a desirable policy — the PPs assigned to local addresses are likely to be overloaded while the remote address PPs remain almost idle.

Perhaps, then, a more dynamic means of allocating incoming messages is needed. But with dynamic allocation of messages to PPs, the PPs will need to communicate and, possibly, synchronize. In essence, the node controller is itself a small multiprocessor, and the PPs will need to coordinate the processing and updating of shared resources such as the directory and local processor cache. Such a situation is likely to introduce many additional protocol race conditions and may limit the latency and occupancy reductions afforded by the multiple protocol processors.

A second option is to try to combine the advantages of a hardwired node controller and a programmable protocol engine while continuing to use the programmable engine for all protocol processing. This approach assumes that a particular protocol (or set of protocols) has been selected and that, as a result, the node controller design can be specialized to the selected protocol. The MAGIC design deliberately avoided protocol-specific optimizations even though protocol-specific hardware or PP instructions could have reduced handler lengths for certain protocols. For example, in a COMA protocol, the length of some protocol handlers can be reduced by adding PP instructions to perform the cache tag matches required to determine whether a line is stored in a node's local memory. In addition to adding protocol-specific PP instructions, generality in the node controller support hardware — MAGIC's jump table in the inbox, for instance — can be reduced, simplifying the design and possibly reducing latency.

Another, perhaps higher performance, node controller design option is to merge a programmable engine into a hardwired node controller architecture. In this approach, a hardwired mechanism would be used to handle the common protocol operations, thereby gaining the lower latencies and occupancies for the vast majority of protocol operations. For protocol corner cases or other non-critical or complex operations, the message instead would be dispatched to a programmable processor core and serviced in a manner similar to that used in MAGIC. The main challenges with this type of hybrid design are what operations to support in the hardwired section, how to decide when to offload an operation to the programmable protocol processor, and how to synchronize and coordinate the hardwired and programmable sections. Ideally, such a design would incorporate a flexible mechanism to allocate an operation to the programmable or hardwired sections. Such flexibility, for example, would allow the programmable engine to be used to service all protocol operations if detailed performance data collection were desired, or to service a selected set of operations in cases where the programmable engine is needed to work around bugs discovered in the hardwired section.

Future work in the design of programmable node controllers could consider issues such as those raised above; indeed, some of the underlying protocol and architecture issues have been explored already in the research community [20, 61, 76, 77, 83]. Clearly, to properly evaluate a potential approach, a much more detailed architectural specification and system simulation is needed to estimate the resulting performance, implementation cost, and implementation complexity.

8.2 Large System Design Issues

Another area of future work is developing the software infrastructure to permit detailed, quantitative protocol and architectural investigations on FLASH. As with any programmable system, much of FLASH's value stems from the software. The MAGIC chip provides a foundation on which various communication paradigms and the protocols to implement them can be studied and compared. To do so requires writing the protocol code, of course, and several protocols have been developed already for FLASH.

Also needed is a comprehensive performance analysis environment that uses the performance monitoring features provided in MAGIC, in the R10000, and in the Spider routers to collect user-specified performance information as an application executes. This performance data can be used to investigate issues such as protocol scalability, operating system scalability and resource management, application performance, and how to best gather, analyze, and report performance data to the application developer and user. MAGIC's flexibility also might be used to aid in implementing a parallel debugging environment or other parallel application development tools.

The FLASH hardware — in particular the capabilities of the MAGIC chip — enables the investigation of many research topics related to the design of next-generation multiprocessors and the operating systems and applications to run on them. Future work in this area is exactly the type of research FLASH is designed to support.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. 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.

[2] A. Agarwal, R. Bianchini, D. Chaiken, et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2-13, June 1995.

[3] T. Agerwala, J. Martin, J. H. Mirza, et al. The SP2 System Architecture. *IBM Systems Journal,* 34(2):152-184, 1995.

[4] G. Alverson, R. Alverson, D. Callahan, et al. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *Proceedings of the ACM International Conference on Supercomputing*, pages 188-197, July 1992.

[5] R. Alverson, D. Callahan, D. Cummings, et al. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1-6, June 1990.

[6] T. E. Anderson, D. E. Culler, D. A. Patterson, et al. A Case for Networks of Workstations: NOW. *IEEE Micro*, 15(1):54-64, February 1995.

[7] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, et al. Empirical Evaluation of the Cray T3D: A Compiler Perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 320-331, June 1995.

[8] G. Astfalk and K. Shaw. Four-State Cache-Coherence in the Convex Exemplar System. Convex Computer Corporation, October 1995.

[9] D. H. Bailey. FFTs in External or Hierarchical Memory. *Journal of Supercomputing*, 4(1):23-35, March 1990.

[10] L. A. Barroso and M. Dubois. The Performance of Cache-Coherent Ring-Based Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 268-277, May 1993.

[11] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, et al. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 3-16, July 1991.

[12] M. Blumrich, K. Li, R. Alpert, et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142-153, April 1994.

[13] N. J. Boden, D. Cohen, R. E. Felderman, et al. Myrinet: A Gigabit Per Second Local Area Network. *IEEE Micro,* 15(1):29-36, February 1995.

[14] J. Boyle, E. Lusk, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, New York, 1987.

[15] A. Brandt. Multi-Level Adaptive Solutions to Boundary-Value Problems. *Mathematics of Computation*, 31(138):333-390, April 1977.

[16] T. Brewer and G. Astfalk. The Evolution of the HP/Convex Exemplar. In *Proceedings of the 42nd COMPCON*, pages 81-86, February 1997.

[17] J. B. Carter et al. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 29(2):219-227, September 1995.

[18] L. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112-1118, December 1978.

[19] D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 314-324, April 1994.

[20] S. Chandra, J. R. Larus, and A. Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61-73, October 1994.

[21] R. Clark. SCI Interconnect Chipset and Adapter: Building Large Scale Enterprise Servers with Pentium Pro SHV Nodes. In *Proceedings of Hot Interconnects IV*, August 1996.

[22] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 98-108, May 1993.

[23] A. L. Cox, S. Dwarkadas, P. Keleher, et al. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 106-117, April 1994.

[24] D. E. Culler, R. M. Karp, D. A. Patterson, et al. LogP: A Practical Model of Parallel Computation. *Communications of the ACM*, 39(11):78-85, November 1996

[25] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, et al. Parallel Computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing*, 1997.

[26] Z. Cvetanovic and D. Bhandarkar. Characterization of Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 60-70, April 1994.

[27] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 2-13, May 1993.

[28] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547-553, May 1987.

[29] W. J. Dally. Wire-efficient VLSI Multiprocessor Communication Networks. In *Proceedings of the 1987 Stanford Conference on Advanced Research in VLSI*, pages 391-415.

[30] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 372-381, April 1996.

[31] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210-220, October 1996.

[32] M. Fillo, S. W. Keckler, W. J. Dally, et al. The M-Machine Multicomputer. *International Journal of Parallel Programming*, 25(3):183-212, June 1997.

[33] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar — A Large Scale Multiprocessor. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 524-529, August 1983.

[34] M. Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, 17(1):34-39, January-February 1997.

[35] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. Ph.D. Dissertation, Stanford University, Stanford, CA, December 1995.

[36] A. Gottlieb, R. Grishman, C. P. Kruskal, et al. The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C-32(2):175-189, February 1983.

[37] A. M. Grizzaffi Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145-156, October 1994.

[38] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794-810, July 1992.

[39] D. B. Gustavson and Q. Li. Local-Area Multiprocessor: The Scalable Coherent Interface. SCIzzL Technical Report, Santa Clara University, 1995.

[40] J. Heinlein, K. Gharachorloo, S. A. Dresser, and A. Gupta. Integration of 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*, pages 38-50, October 1994.

[41] J. Heinlein, R. P. Bosch, and K. Gharachorloo. Coherent Block Data Transfer in the FLASH Multiprocessor. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 18-27, April 1997.

[42] J. Heinlein. Personal communication, October 1997.

[43] J. Heinlein. *Optimized Multiprocessor Communication and Synchronization Using a Programmable Protocol Engine*. Ph.D. Dissertation, Stanford University, Stanford, CA, *to appear.*

[44] M. Heinrich, J. Kuskin, D. Ofelt, et al. 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*, pages 274-285, October 1994.

[45] M. Heinrich. *The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols*. Ph.D. Dissertation, Stanford University, Stanford, CA, *to appear.*

[46] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan Kaufmann, San Francisco, CA, 1996.

[47] W. D. Hillis and L. W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11):30-40, November 1993.

[48] C. Holt and J. P. 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.

[49] C. Holt, J. P. Singh, and J. L. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 134-145, May 1996.

[50] C. Holt, M. Heinrich, and J. P. Singh. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[51] C. Hristea. Measuring CC-MP Memory Hierarchy Performance Using Microbenchmarks. Presentation to Scalable Systems Group at Silicon Graphics, January 1997.

[52] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 122-133, May 1996.

[53] Intel Corporation, Supercomputer Systems Division. Paragon Technical Summary, 1993.

[54] Intel Corporation. Intel's ASCI Red Project Hits 200 GFLOPS Milestone. Press release at <http://www.ssd.intel.com>, November 1996.

[55] T. Joe and J. L. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 82-93, April 1994.

[56] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. *Operating Systems Review*, 29(5):213-228, December 1995.

[57] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[58] M. Kantrowitz and L. M. Noack. Functional Verification of a Multiple-Issue, Pipelined, Superscalar Alpha Processor — the Alpha 21164 CPU Chip. *Digital Technical Journal*, 7(1):136-144, 1995.

[59] V. Karamcheti and A. A. Chien. A Comparison of Architectural Support for Messaging on the TMC CM-5 and Cray T3D. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 298-307, June 1995.

[60] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.In *Proceedings of the Winter 1994 USENIX Conference*, pages 115-32, January 1994.

[61] A. Krishnamurthy, K. E. Schauser, C. J. Scheiman, et al. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37-48, October 1996.

[62] J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April 1994.

[63] J. Kuskin. *MAGIC Architecture and Design Specification*. Technical report, Computer Systems Laboratory, Stanford University, *to appear.*

[64] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241-251, June 1997.

[65] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. Ph.D. Dissertation, Stanford University, Stanford, CA, December 1991.

[66] D. Lenoski, J, Laudon, K. Gharachorloo, et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63-79, March 1992.

[67] D. Lenoski, J. Laudon, T. Joe, et al. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[68] T. D. Lovett, R. M. Clapp, and R. J. Safranek. NUMA-Q: An SCI-Based Enterprise Server. Sequent Computer Corporation, 1996.

[69] T. D. Lovett and R. M. 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.

[70] LSI Logic Corporation. LCB500K Standard Cell Databook, 1996.

[71] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 85-97, June 1997.

[72] M. R. Martonosi. *Analyzing and Tuning Memory Performance in Sequential and Parallel Programs*. Ph.D. Dissertation, Stanford University, Stanford, CA, January 1994.

[73] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. In *Proceedings of SIGMETRICS 1996*, May 1996.

[74] M. M. Michael, A. K. Nanda, B.-H. Lim, and M. L. Scott. Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 219-228, June 1997.

[75] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. Ph.D. Dissertation, Stanford University, Stanford, CA, June 1994.

[76] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 247-258, May 1996.

[77] S. S. Mukherjee and M. D. Hill. A Survey of User-Level Network Interfaces for System Area Networks. Technical Report 1340, Computer Sciences Department, University of Wisconsin-Madison, February 1997.

[78] S. S. Mukherjee and M. D. Hill. A Case for Making Network Interfaces Less Peripheral. In *Proceedings of Hot Interconnects V,* August 1997.

[79] A. Nowatzyk, G. Aybay, and M. Browne. The S3.mp Scalable Shared Memory Multiprocessor. In Proceedings of 24th International Conference on Parallel Processing, pages 1-10, August 1995.

[80] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224-235, May 1993.

[81] PCI Special Interest Group. PCI Local Bus Specification, Revision 2.1. June, 1995.

[82] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325-337, April 1994.

[83] S. K. Reinhardt, M. D. Hill, J. R. Larus, et al. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Performance Evaluation Review*, 21:(1):48-60, June 1993.

[84] S. K. Reinhardt, R. A. Pfile, and D. A. Wood. T-Zero: Hardware Support for Distributed Shared Memory on a Cluster of Workstations. In *Proceedings of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[85] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. *In Proceedings of the 23rd International Symposium on Computer Architecture*, pages 34-43, May 1996.

[86] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4):34-43, Winter 1995.

[87] M. Rosenblum, J. Chapin, D. Teodosiu, et al. Implementing Efficient Fault Containment for Multiprocessors. *Communications of the ACM*, 39(3):52-61, September 1996.

[88] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modeling and Computer Simulation*, pages 78-103, January 1997.

[89] E. Rothberg, J. P. Singh, and A. Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 14-25, May 1993.

[90] A. Saulsbury and A. Nowatzyk. Simple COMA on S3.mp. Presentation at The Fifth Workshop on Scalable Shared Memory Multiprocessors, Santa Margherita Ligure, Italy, June 1995.

[91] Scalable Coherent Interface, ANSI/IEEE Standard 1596-1992, August 1993.

[92] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174-185, October 1996.

[93] I. Schoinas, B. Falsafi, A. A. Lebeck, et al. 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 297-307, October 1994.

[94] S. 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, October 1996.

[95] R. Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. Ph.D. Dissertation, Stanford University, Stanford, CA, October 1992.

[96] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, 26(7):42-50, July 1993.

[97] J. P. Singh, T. Joe, A. Gupta, and J. L. Hennessy. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. In *Proceedings of Supercomputing '93*, pages 214-225, November 1993.

[98] M. D. Smith. *Support for Speculative Execution in High-Performance Processors*. Ph.D. Dissertation, Stanford University, Stanford, CA, November 1992.

[99] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea. The Communication Software and Parallel Environment of the IBM SP2. *IBM Systems Journal*, 34(2):205-221, 1995.

[100] E. Spertus, S. C. Goldstein, K. E. Schauser, et al. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and CM-5. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 302-313, May 1993.

[101] R. Stallman. Using and Porting GNU CC. Free Software Foundation, Cambridge MA, January 1992.

[102] C. B. Stunkel, D. G. Shea, B. Aball, et al. The SP2 High-Performance Switch. *IBM Systems Journal*, 34(2):185-204, 1995.

[103] Tera Computer Company. Tera Computer Company Announces Booting Of Operating System. Press release, August 1997.

[104] J. Torrellas. *Multiprocessor Cache Memory Performance: Characterization and Optimization*. Ph.D. Dissertation, Stanford University, Stanford, CA, August 1992.

[105] T. von Eicken, D. E. Culler, S. C. Goldstein, et al. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256-266, May 1992.

[106] D. W. Walker and J. J. Dongarra. MPI: A Standard Message Passing Interface. *Supercomputer*, 12(1):56-68 January 1996.

[107] W.-D. Weber, S. Gold, P. Helland, et al. 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, June 1997.

[108] W.-D. Weber. *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*. Ph.D. Dissertation, Stanford University, Stanford, CA, January 1993.

[109] S. C. Woo, J. P. Singh, and J. L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219-229, October 1994.

[110] S. C. Woo, M. Ohara, E. Torrie, et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24-36, June 1995.

[111] S. C. Woo. *The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors*. Ph.D. Dissertation, Stanford University, Stanford, CA, June 1996.

[112] D. A. Wood, S. Chandra, B. Falsafi, et al. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 156-167, May 1993.

[113] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28-40, April 1996.

[114] W. C. Yen, D. W. Yen, and K.-S. Fu. Data Coherence Problem in a Multicache System. *IEEE Transactions on Computers*, C-34(1):56-65, January 1985.

[115] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 44-55, May 1996.