The Stanford FLASH Multiprocessor

Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein,
Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter,
Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

The FLASH multiprocessor efficiently integrates support for 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 a custom node controller called MAGIC. 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 use of the protocol processor makes FLASH very flexible — it can support a variety of different communication mechanisms — and simplifies the design and implementation.

This paper presents the architecture of FLASH and MAGIC, and discusses the base cache-coherence and message-passing protocols. Latency and occupancy numbers, which are derived from our system-level simulator and our Verilog code, are given for several common protocol operations. The paper also describes our software strategy and FLASH's current status.

1 Introduction

The two architectural techniques for communicating data among processors in a scalable multiprocessor are message passing and distributed shared memory (DSM). Despite significant differences in how programmers view these two architectural models, the underlying hardware mechanisms used to implement these approaches have been converging. Current DSM and message-passing multiprocessors consist of processing nodes interconnected with a high-bandwidth network. Each node contains a node processor, a portion of the physically distributed memory, and a node controller that connects the processor, memory, and network together. The principal difference between message-passing and DSM machines is in the protocol implemented by the node controller for transferring data both within and among nodes.

Perhaps more surprising than the similarity of the overall structure of these types of machines is the commonality in functions performed by the node controller. In both cases, the primary performance-critical function of the node controller is the movement of data at high bandwidth and low latency among the processor, memory, and network. In addition to these existing similarities, the architectural trends for both styles of machine favor further convergence in both the hardware and software mechanisms used to implement the communication abstractions. Message-passing machines are moving to efficient support of short messages and a uniform address space, features normally associated with DSM machines. Similarly, DSM machines are starting to provide support for message-like block transfers (e.g., the Cray T3D), a feature normally associated with message-passing machines.

The efficient integration and support of both cache-coherent shared memory and low-overhead user-level message passing is the primary goal of the FLASH (FLexible Architecture for SHared memory) multiprocessor. Efficiency involves both low hardware overhead and high performance. A major problem of current cache-coherent DSM machines (such as the earlier DASH machine [LLG+92]) is their high hardware overhead, while a major criticism of current message-passing machines is their high software overhead for user-level message passing. FLASH integrates and streamlines the hardware primitives needed to provide low-cost and high-performance support for global cache coherence and message passing. We aim to achieve this support without compromising the protection model or the ability of an operating system to control resource usage. The latter point is important since we want FLASH to operate well in a general-purpose multiprogrammed environment with many users sharing the machine as well as in a traditional supercomputer environment.

To accomplish these goals we are designing a custom node controller. This controller, called MAGIC (Memory And General Interconnect Controller), is a highly integrated chip that implements all data transfers both within

1063-6897/94 $03.00 © 1994 IEEE

Petition for *Inter Partes* Review of
U.S. Pat. No. 7,296,121
IPR2015-00158

1006

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

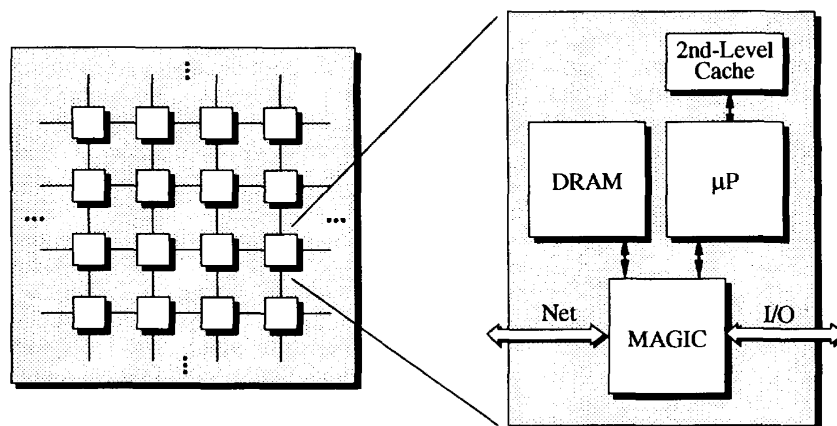

Figure 2.1. FLASH system architecture.

the node and between the node and the network. To deliver high performance, the MAGIC chip contains a specialized data path optimized to move data between the memory, network, processor, and I/O ports in a pipelined fashion without redundant copying. To provide the flexible control needed to support a variety of DSM and message-passing protocols, the MAGIC chip contains an embedded processor that controls the data path and implements the protocol. The separate data path allows the processor to update the protocol data structures (e.g., the directory for cache coherence) in parallel with the associated data transfers.

This paper describes the FLASH design and rationale. Section 2 gives an overview of FLASH. Section 3 briefly describes two example protocols, one for cache-coherent shared memory and one for message passing. Section 4 presents the microarchitecture of the MAGIC chip. Section 5 briefly presents our system software strategy and Section 6 presents our implementation strategy and current status. Section 7 discusses related work and we conclude in Section 8.

2 FLASH Architecture Overview

FLASH is a single-address-space machine consisting of a large number of processing nodes connected by a low-latency, high-bandwidth interconnection network. Every node is identical (see Figure 2.1), containing a high-performance off-the-shelf microprocessor with its caches, a portion of the machine's distributed main memory, and the MAGIC node controller chip. The MAGIC chip forms the heart of the node, integrating the memory controller, I/O controller, network interface, and a programmable protocol processor. This integration allows for low hardware overhead while supporting both cache-coherence and message-passing protocols in a scalable and cohesive fashion.[1]

The MAGIC architecture is designed to offer both flexibility and high performance. First, MAGIC includes a programmable protocol processor for flexibility. Second, 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. Third, to avoid limiting the node design to any specific protocol and to accommodate protocols with varying memory requirements, the 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 contains on-chip instruction and data caches. Finally, MAGIC separates data movement logic from protocol state manipulation logic. The 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 employs 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.

FLASH nodes communicate by sending intra- and inter-node commands, which we refer to as *messages*. To implement a protocol on FLASH, one must define what kinds of messages will be exchanged (the *message types*),

1. Our decision to use only one compute processor per node rather than multiple processors was driven mainly by pragmatic concerns. Using only one processor considerably simplifies the node design, and given the high bandwidth requirements of modern processors, it was not clear that we could support multiple processors productively. However, nothing in our approach precludes the use of multiple processors per node.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

and write the corresponding code sequences for the protocol processor (the *handlers*). Each handler performs the necessary actions based on the machine state and the information in the message it receives. Handler actions include updating machine state, communicating with the local 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.

Since FLASH is designed to scale to thousands of processing nodes, a comprehensive protection and fault containment strategy is needed to assure acceptable system availability. At the user level, the virtual memory system provides protection against application software errors. However, system-level errors such as operating system bugs and hardware faults require a separate fault detection and containment mechanism. The hardware and operating system cooperate to identify, isolate, and contain these faults. MAGIC provides a hardware-based "firewall" mechanism that can be used to prevent certain operations (memory writes, for example) from occurring on unauthorized addresses. Error-detection codes ensure data integrity: ECC protects main memory and CRCs protect network traffic. Errors are reported to the operating system, which is responsible for taking suitable action.

3 FLASH Protocols

This section presents a base cache-coherence protocol and a base block-transfer protocol we have designed for FLASH. We use the term "base" to emphasize that these two protocols are simply the ones we chose to implement first; Section 3.3 discusses protocol extensions and alternatives.

3.1 Cache Coherence Protocol

The base cache-coherence protocol is directory-based and has two components: a scalable directory data structure, and a set of handlers. For a scalable directory structure, FLASH uses *dynamic pointer allocation* [Simoni92], illustrated in Figure 3.1. In this scheme, each cache line-sized block — 128 bytes in the prototype — of main memory is associated with an 8-byte state word called a *directory header*, which is stored in a contiguous section of main memory devoted solely to the cache-coherence protocol. Each directory header contains some boolean flags and a link field that points to a linked list of sharers. For efficiency, the first element of the sharer list is stored in the directory header itself. If a block of memory is cached by more than one processor, additional memory for its list of sharers is allocated from the *pointer/link store*. Like the directory headers, the pointer/link store is also a physically contiguous region of main memory. Each entry in the pointer/link store consists of a pointer to the sharing processor, a link to the next entry in the list, and an end-of-list bit. A free list is used to track the available entries in the pointer/link store. Pointer/link store entries are allocated from the free list as cache misses are satisfied, and are returned to the free list either when the line is written and invalidations are sent to each cache on the list of sharers, or when a processor notifies the directory that it is no longer caching a block[2].

A significant advantage of dynamic pointer allocation is that the directory storage requirements are scalable. The amount of memory needed for the directory headers is proportional to the local memory per node, and scales as more processors are added. The total amount of memory needed in the machine for the pointer/link store is proportional to the total amount of cache in the system. Since the amount of cache is much smaller that the amount of main memory, the size of the pointer/link store is sufficient to maintain full caching information, as long as the loading on the different memory modules is uniform. When this uniformity does not exist, a node can run out of pointer/link storage. While a detailed discussion is beyond the scope of this paper, several heuristics can be used in this situation to ensure reasonable performance. Overall, the directory occupies 7% to 9% of main memory, depending on system configuration.

Apart from the data structures used to maintain directory information, the base cache-coherence protocol is similar to the DASH protocol [LLG+90]. Both protocols utilize separate request and reply networks to eliminate request-reply cycles in the network. Both protocols forward dirty data from a processor's cache directly to a requesting processor, and both protocols use negative acknowledgments to avoid deadlock and to cause retries when a requested line is in a transient state. The main difference between the two protocols is that in DASH each cluster collects its own invalidation acknowledgments, whereas in FLASH invalidation acknowledgments are col-

2. The base cache-coherence protocol relies on replacement hints. The protocol could be modified to accommodate processors which do not provide these hints.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

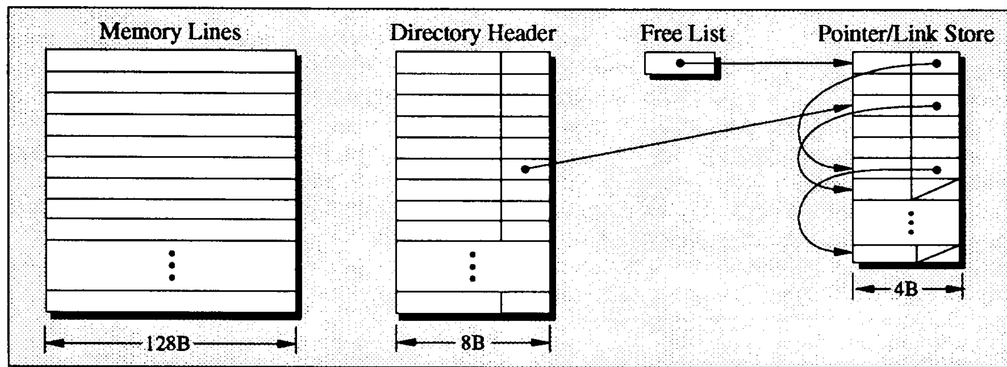

Figure 3.1. Data structures for the dynamic pointer allocation directory scheme.

lected at the *home* node, that is, the node where the directory data is stored for that block.

Avoiding deadlock is difficult in any cache-coherence protocol. Below we discuss how the base protocol handles the deadlock problem, and illustrate some of the protocol-independent deadlock avoidance mechanisms of the MAGIC architecture. Although this discussion focuses on the base cache-coherence protocol, any protocol run on FLASH can use these mechanisms to eliminate the deadlock problem.

As a first step, the base protocol divides all messages into requests (e.g., read, read-exclusive, and invalidate requests) and replies (e.g., read and read-exclusive data replies, and invalidation acknowledgments). Second, the protocol uses the virtual lane support in the network routers to transmit requests and replies over separate logical networks. Next, it guarantees that replies can be *sunk*, that is, replies generate no additional outgoing messages. This eliminates the possibility of request-reply circular dependencies. To break request-request cycles, requests that cannot be sunk may be negatively acknowledged, effectively turning those requests into replies.

The final requirement for a deadlock solution is a restriction placed on all handlers: they must yield the protocol processor if they cannot run to completion. If a handler violates this constraint and stalls waiting for space on one of its output queues, the machine could potentially deadlock because it is no longer servicing messages from the network. To avoid this type of deadlock, the scheduling mechanism for the incoming queues is initialized to indicate which incoming queues contain messages that may require outgoing queue space. The scheduler will not select an incoming queue unless the corresponding outgoing queue space requirements are satisfied.

However, in some cases, the number of outgoing messages a handler will send cannot be determined beforehand, preventing the scheduler from ensuring adequate outgoing queue space for these handlers. For example, an incoming request (for which only outgoing reply queue space is guaranteed) may need to be forwarded to a dirty remote node. If at this point the outgoing request queue is full, the protocol processor negatively acknowledges the incoming request, converting it into a reply. A second case not handled by the scheduler is an incoming write miss that is scheduled and finds that it needs to send N invalidation requests into the network. Unfortunately, the outgoing request queue may have fewer than N spots available. As stated above, the handler cannot simply wait for space to free up in the outgoing request queue to send the remaining invalidations. To solve this problem, the protocol employs the *software queue* where it can suspend messages to be rescheduled at a later time.

The software queue is a reserved region of main memory that any protocol can use to suspend message processing temporarily. For instance, each time MAGIC receives a write request to a shared line, the corresponding handler reserves space in the software queue for possible rescheduling. If the queue is already full, the incoming request is simply negatively acknowledged. This case should be extremely rare. If the handler discovers that it needs to send N invalidations, but only $M < N$ spots are available in the outgoing request queue, the handler sends M invalidate requests and then places itself on the software queue. The list of sharers at this point contains only those processors that have not been invalidated. When the write request is rescheduled off of the software queue, the new handler invocation continues sending invalidation requests where the old one left off.

3.2 Message Passing Protocol

In FLASH, we distinguish long messages, used for block transfer, from short messages, such as those required

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

for synchronization. This section discusses the block transfer mechanism; Section 3.3 discusses short messages.

The design of the block transfer protocol was driven by three main goals: provide user-level access to block transfer without sacrificing protection; achieve transfer bandwidth and latency comparable to a message-passing machine containing dedicated hardware support for this task; and operate in harmony with other key attributes of the machine including cache coherence, virtual memory, and multiprogramming [HGG94]. We achieve high performance because MAGIC efficiently streams data to the receiver. The performance is further improved by the elimination of processor interrupts and system calls in the common case, and by the avoidance of extra copying of message data.

To distinguish a user-level message from the low-level messages MAGIC sends between nodes, this section explicitly refers to the former as a *user message*. Sending a user message in FLASH logically consists of three phases: initiation, transfer, and reception/completion.

To send a user message, an application process calls a library routine to communicate the parameters of the user-level message to MAGIC. This communication happens using a series of uncached writes to special addresses (which act as memory-mapped commands). Unlike standard uncached writes, these special writes invoke a different handler that accumulates information from the command into a message description record in MAGIC's memory. The final command is an uncached read, to which MAGIC replies with a value indicating if the message is accepted. Once the message is accepted, MAGIC invokes a *transfer handler* that takes over responsibility for transferring the user message to its destination, allowing the main processor to run in parallel with the message transfer.

The transfer handler sends the user message data as a series of independent, cache line-sized messages. The transfer handler keeps the user message data coherent by checking the directory state as the transfer proceeds, taking appropriate coherence actions as needed. Block transfers are broken into cache line-sized chunks because the system is optimized for data transfers of this size, and because block transfers can then utilize the deadlock prevention mechanisms implemented for the base cache-coherence protocol. From a deadlock avoidance perspective, the user message transfer is similar to sending a long list of invalidations: the transfer handler may only be able to send part of the user message in a single activation. To avoid filling the outgoing queue and to allow other handlers to execute, the transfer handler periodically marks its progress and suspends itself on the software queue.

When each component of the user-level message arrives at the destination node, a *reception handler* is invoked which stores the associated message data in memory and updates the number of message components received. Using information provided in advance by the receiving process, the handler can store the data directly in the user process's memory without extra copying. When all the user message data has been received, the handler notifies the local processor that a user message has arrived (the application can choose to poll for the user message arrival or be interrupted), and sends a single acknowledgment back to the sender, completing the transfer.

Section 4.3 discusses the anticipated performance of this protocol.

3.3 Protocol Extensions and Alternatives

MAGIC's flexible design supports a variety of protocols, not just the two described in Section 3.1 and Section 3.2. By changing the handlers, one can implement other cache-coherence and message-passing protocols, or support completely different operations and communication models. Consequently, FLASH is ideal for experimenting with new protocols.

For example, the handlers can be modified to emulate the "attraction memory" found in a cache-only memory architecture, such as Kendall Square Research's ALLCACHE [KSR92]. A handler that normally forwards a remote request to the home node in the base cache-coherence protocol can be expanded to first check the local memory for the presence of the data. Because MAGIC stores protocol data structures in main memory, it has no difficulty accommodating the different state information (e.g., attraction memory tags) maintained by a COMA protocol.

Another possibility is to implement synchronization primitives as MAGIC handlers. Primitives executing on MAGIC avoid the cost of interrupting the main processor and can exploit MAGIC's ability to communicate efficiently with other nodes. In addition, guaranteeing the atomicity of the primitives is simplified since MAGIC handlers are non-interruptible. Operations such as fetch-and-op and tree barriers are ideal candidates for this type of implementation.

FLASH's short message support corresponds closely to the structuring of communication using active messages as advocated by von Eicken et al. [vECG+92]. However, the MAGIC chip supports fast active messages only at the system level, as opposed to the user level. While von Eicken et al. argue for user-level active messages, we have found that system-level active messages suffice and in many ways simplify matters. For example, consider the shared-memory model and the ordinary read/write requests issued by compute processors. Since the virtual addresses issued by the processor are translated into physical addresses and are protection-checked by the TLB before they reach the MAGIC chip, no further translation or protection checks

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming inter-face) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.