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

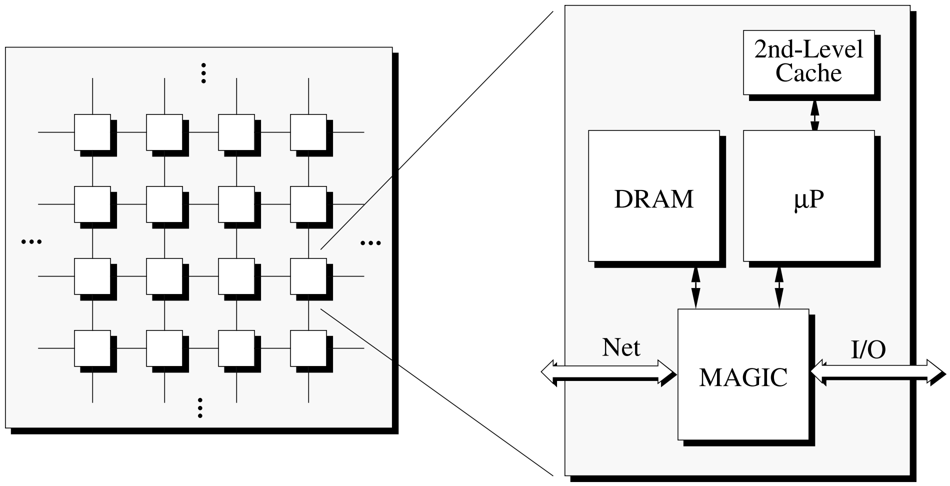

Figure 2.1. FLASH system architecture.

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.

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.

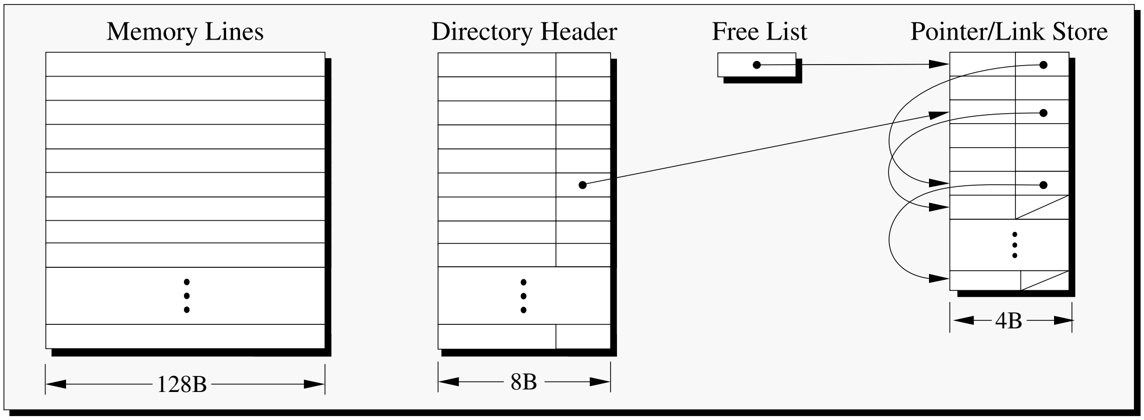

Figure 3.1. Data structures for the dynamic pointer allocation directory scheme.

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 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 are needed at MAGIC. By not allowing user-level handlers, we ensure that malicious user-level handlers do not cause deadlock by breaking resource consumption conventions in the MAGIC chip. The MAGIC chip architecture could be extended to provide protection for user-level handlers (e.g., by providing time-outs), but this change would significantly complicate the chip and the protocols. Instead, we are investigating software techniques for achieving the required protection to allow user-level handlers to execute in the unprotected MAGIC environment. Overall, we believe the disadvantages of providing hardware support for user-level handlers in MAGIC outweigh the advantages. Operations that are truly critical to performance (e.g., support for tree barriers and other synchronization primitives) usually can be coded and provided at the system level by MAGIC. Disallowing user-level handlers should lead to a simpler and higher-performing design.

While the inherent complexity of writing handlers may be small, it is important to realize that errors in the MAGIC handlers directly impact the correctness and stability of the machine. We consider the verification of handlers to be analogous to hardware verification, since MAGIC handlers directly control the node's hardware resources. As a result, although new protocols may be simple to implement, they must be verified thoroughly to be trusted to run on MAGIC.

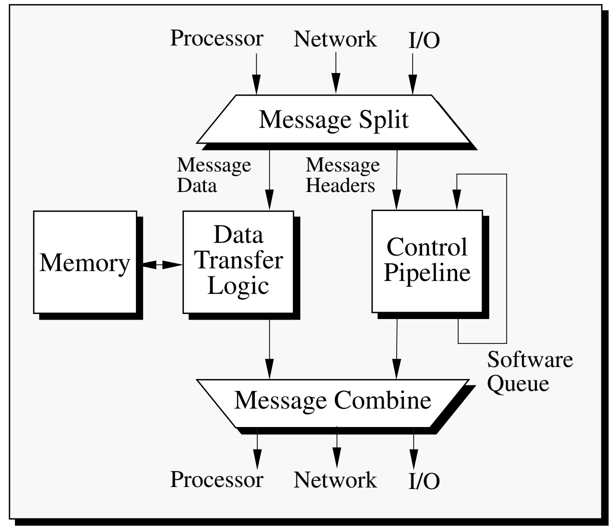

Figure 4.1. Message flow in MAGIC.

4 MAGIC Microarchitecture

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. 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 are merged with the associated outgoing message data to form complete outgoing messages.

4.1 The 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 (I/O) 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 storage 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

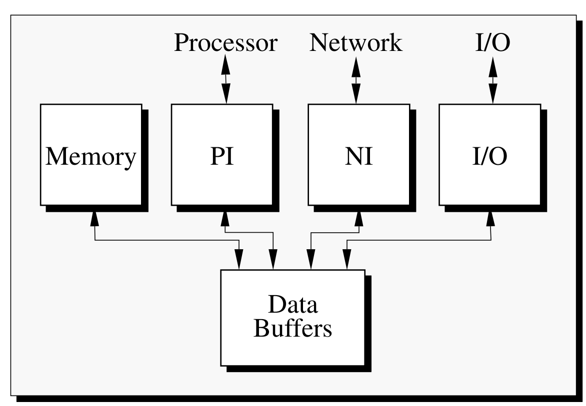

Figure 4.2. Data transfer logic.

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 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 cause the data transfer logic to deliver the data to the proper destination.

4.2 The 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 I/O 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 I/O 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.

4.2.1 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 I/O 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 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 discussed in Section 3.1. Each incoming queue has an associated array of status bits which the PP can use to specify the queue's relative priority, 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,

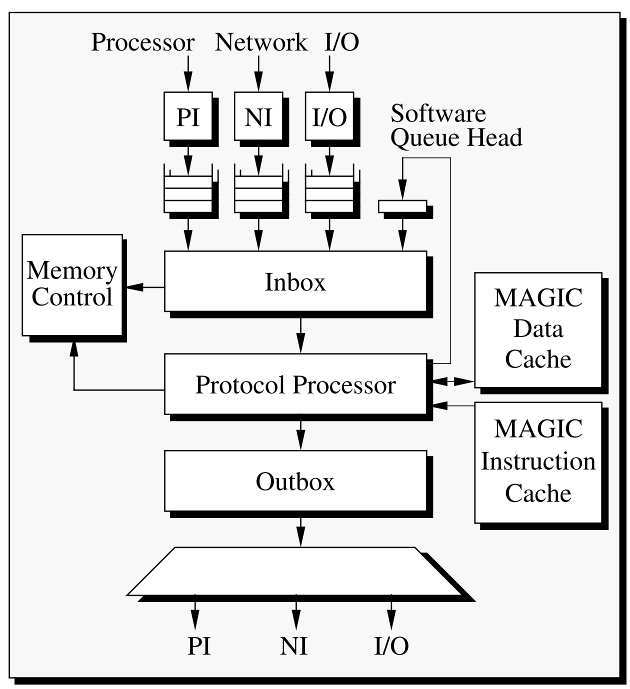

Figure 4.3. Control macropipeline.

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.

4.2.2 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 DLX instruction set [HP90] 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, exceptions, pipeline interlocks, or hardware address translation. Although these features can simplify some aspects of protocol processing, we elected to eliminate them 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 pipeline interlocks falls on the PP programmer or compiler.

4.2.3 Outbox Operation

The outbox performs outgoing message sends on behalf of the PP. It provides a high-level interface to the PI, NI, and I/O, 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 I/O, 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.

4.2.4 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 I/O — implement these interfaces, isolating the rest of the chip from the interface details. This isolation is another component of MAGIC's flexibility, since it limits the amount of hardware modifications required to adapt the MAGIC design for use in other systems.

4.3 Putting It All Together

To make the discussion in the previous sections more concrete, this section summarizes the interaction of the data transfer logic and the control macropipeline and presents cycle counts for common protocol operations.

To achieve system performance competitive with a fully-hardwired design, MAGIC must minimize the latency required to service requests from the main processor. Minimizing main processor request latency requires MAGIC, internally, to minimize the data transfer logic time and, at the same time, ensure that the control macropipeline time is less than the data transfer time. Overall, MAGIC's performance can be measured both by the total time to process a message (the latency) and by the rate at which sustained message processing can occur (the occupancy).

To demonstrate that MAGIC can achieve competitive performance, we present the latency for servicing a processor read miss to local memory. Table 4.1 lists the latency through each stage of the data transfer logic and control macropipeline for this operation, assuming that the MAGIC chip was initially idle. The cycle counts are based on a 100 MHz (10 ns) MAGIC clock rate and are derived from the current Verilog models of the various units.

One cycle after the miss appears on the processor bus, the PI places the message in its incoming queue. After three additional cycles, the inbox has read the message from the PI queue, passed the message header through the jump table, and initiated a speculative memory read. At this point, state manipulation and data transfer proceed in parallel. The PP requires 10 cycles to update the directory information and forward the data to the processor (assuming all accesses hit in the MAGIC caches). The memory system returns the first 8 bytes of data after 16 cycles and the remaining 120 bytes in the next 15 cycles. Four cycles after the first data bytes return from the memory system, the PI issues the response on the processor bus and begins delivering the cache fill data to the processor.

Table 4.1. Cycle Counts for Local Read Miss

Unit	Control Macropipeline Latency, 10 ns Cycles	Data Transfer Pipeline Latency, 10 ns Cycles
PI, incoming	1	1
Inbox, schedule and jump table lookup	3	3
PP	10	—
Outbox	1	—
PI, outgoing	4	4
Memory read, time to first 8 bytes	—	16
Memory read, time for remaining 120 bytes	—	15
TOTAL	19	24 to first word, 39 total

This example illustrates several important MAGIC architectural features. First, the speculative memory indication contained in the jump table allows the memory read to be started before the PP begins processing the message, thereby reducing the cache fill time. Second, the data transfer proceeds independently of the control processing; thus, at the same time the memory system is reading the data from memory into the data buffer, the PP is updating directory state and composing the outgoing data reply message. Third, the valid bits associated with the data buffer allow the PI to pipeline the data back to the processor, further reducing the cache fill time. Fourth, the separation of data and control processing eliminates multiple data copies: the data flows, via the data buffer, directly from the memory system to the PI while the control processing flows through the entire control macropipeline. Finally, as Table 4.1 demonstrates, the control macropipeline time is less than the data transfer time, indicating that the flexible protocol processing is not the limiting factor of MAGIC's performance.

The previous example focused on the latency required to service the processor's cache fill request. As noted above, the occupancy of message processing is also important. The occupancy is determined by the longest suboperation that must be performed. Again, competitive performance requires that the control macropipeline not be the longest suboperation. Table 4.2 lists PP occupancy cycle counts for some of the common handlers in the base cache-coherence protocol; the occupancies of the other units are insensitive to the type of message being processed. (Of course the data transfer logic time is eliminated for messages that do not require data movement.) The first three entries are operations that require local memory access; the final two entries require the PP to pass a message between the network and local processor. For all operations with a local memory access, the control macropipeline time is less than the data transfer time. Hence, as would be the case in a fully-hardwired controller, the sustained rate at which MAGIC can supply data is governed by the memory system.

Table 4.2. PP Occupancies For Common Handlers

Handler	Occupancy (cycles)
Local write miss (no shared copies)	10
Local write miss (shared copies)	7 + 13 per invalidation
Remote read miss, clean	14
Outgoing pass through on remote miss	3
Incoming pass through on remote miss	3

The PP cycle counts for the operations in Table 4.1 and Table 4.2 assume that all PP loads and stores hit in the on-chip MAGIC data cache. Our initial studies, based on address traces collected from the DASH machine and on small executions of SPLASH [SWG92] applications, show that the PP reference stream has sufficient locality to make this a reasonable assumption.

We have coded a C version of the block-transfer protocol in our system-level simulator and, for performance studies, have hand-coded some of the critical handlers in PP assembly language. Our preliminary studies, based on these handlers, show that the initiation phase of the protocol takes approximately 70 cycles until the processor can continue from a non-blocking send. After performing transfer setup for 30 additional cycles, the PP begins the user message transfer. Once the transfer has started, MAGIC is capable of transferring user data at a sustained rate of one 128-byte cache line every 30 – 40 cycles, yielding a useful bandwidth of 300 – 400 MB/s.

5 System Software Strategy

As noted in Section 1, our goal is for FLASH to operate well both in a traditional supercomputer environment and in a general-purpose, multiprogrammed environment. The latter environment poses significant challenges since general-purpose environments typically contain large numbers of processes making many system calls and small I/O requests. In addition, users expect good interactive response, fair sharing of the hardware, protection against malicious programs, and extremely rare system crashes. The combination of workload characteristics and user requirements rules out using a standard supercomputer operating system for FLASH. Scalability and fault containment requirements rule out using a general-purpose operating system designed for small-scale shared-memory multiprocessors. To address these issues, we are designing a new operating system for FLASH, called *Hive*.

Beneath a standard shared-memory multiprocessor API, Hive organizes the hardware nodes into groups called *cells.* Each cell runs a semi-autonomous NUMA-aware operating system. The division into cells provides the replication needed for a highly-scalable operating system implementation and allows for fault containment. Cells interact as a distributed system internal to Hive, using the firewall and message-passing mechanisms of the MAGIC chip for fault isolation. The number of nodes per cell can be configured for different workload and availability requirements.

Hive supports shared memory between applications on different cells, and can allocate processor and memory resources from multiple cells to a single application. This support allows flexible, fine-grained sharing of the machine by large and small applications, despite the fault containment partitions. A more detailed discussion of Hive's design is beyond the scope of this paper.

6 Implementation and Status

FLASH will use the MIPS T5, a follow-on to the R4000, as its primary processor. Like the R4000, the T5 manages its own second-level cache. The target speed of the node board and the MAGIC chip is 100 MHz. The multiply-banked memory system is designed to match the node's bandwidth requirements and is optimized for 128-byte transfers, the system cache line size. FLASH will implement the PCI standard bus for its I/O subsystem and will use next-generation Intel routers for the interconnection network. The initial FLASH prototype will contain 256 processing nodes. We plan to collaborate with the Intel Corporation and Silicon Graphics on the design and construction of the prototype machine.

We currently have a detailed system-level simulator up and running. The simulator is written in C++ as a multithreaded memory simulator for Tango-Lite [Golds93]. The entire system, called FlashLite, runs real applications and enables us to verify protocols, analyze system performance, and identify architectural bottlenecks. We have coded the entire base cache-coherence and base block-transfer protocols for FlashLite, and have run complete simulations of several SPLASH applications. The FlashLite code is structured identically to the actual hardware, with each hardware block corresponding to a FlashLite thread. To aid the debugging of protocols implemented in FlashLite we have developed a random test case generator, the FLASH Protocol Verifier.

On the hardware design front we are busily coding the Verilog description of the MAGIC chip. To verify our hardware description we plan to have FlashLite provide test vectors for a Verilog run, and to run real N processor applications with N–1 FlashLite nodes and one Verilog node. Since the FlashLite code is structured like the Verilog description, we also plan to replace a single FlashLite thread with the appropriate hardware block description to allow more efficient and accurate verification.

Software tools for the protocol processor are another major effort. We are porting the GNU C compiler [Stall93] to generate code for the 64-bit PP. We have also ported a superscalar instruction scheduler and an assembler from the Torch project [SHL92]. Finally, we have a PP instruction set emulator ported from Mable. This emulator will help us verify the actual PP code sequences by becoming the PP thread in FlashLite simulations.

Operating system development is proceeding concurrently with the hardware design. Hive's implementation is based on IRIX (UNIX SVR4 from Silicon Graphics), with extensive modifications in progress to the virtual memory, I/O, and process management subsystems.

7 Related Work

The architecture of FLASH builds on the work of many previous research projects. In this section we compare FLASH to several existing and proposed machines.

The Alewife [ACD+91, KA93] machine from MIT shares with FLASH the goals of providing a cache-coherent single-address-space machine with integrated message passing. It is also similar to FLASH in that the directory information for coherence is kept in a reserved portion of the distributed main memory. However, it is different in that the common cases of the coherence protocol are fixed in hardware finite state machines, and therefore the base protocol cannot be changed significantly. The Alewife machine also does not support virtual memory, and as a result, many of the issues that arise in doing user-level protected messaging in the presence of multiprogramming are currently not addressed. FLASH directly addresses these issues.

The J-machine project at MIT [NWD93, SGS+93] has also focused on supporting multiple communication paradigms within a single machine. A key difference in the approaches, however, is that while the J-machine uses the same processor for both computing and protocol processing, FLASH uses separate compute and protocol processors. Although a single processor obviously costs less, our experiences with DASH in supporting cache coherence suggest that, if the protocol and compute processor were one and the same, the sheer number of protocol processing requests would cause the compute performance to degrade considerably. Furthermore, we believe that the task requirements for the compute processor and the protocol processor are fundamentally different, and therefore the two components need to be architected differently, especially if one relies on an off-the-shelf compute processor. For instance, our compute processor is designed for throughput on code optimized for locality, while the protocol processor is optimized to process many very short handler sequences, with successive sequences bearing little relationship to each other. Another difference is that whenever the compute processor makes a memory request, it does so with the intention of using the data returned for further computation. In contrast, when the protocol processor makes a memory request, it directly passes the data returned by memory to the network or to the local processor; the data need never go to the protocol processor's registers or through its memory hierarchy. This data handling method leads to significant differences in the capabilities of FLASH and the J-machine. For example, the J-machine processor, the MDP, has neither the throughput of our compute processor (e.g., it does not have caches or significant number of registers), nor does it have the extra hardware support for data transfer provided in MAGIC to handle remote requests efficiently.

The Thinking Machine's CM-5 [TMC92] provides user-level access to the network so that short user messages can be sent efficiently by the source processor. However, in contrast to FLASH, the main processor is involved in all transactions on both the sending and receiving ends. As discussed for the J-machine, we believe this can considerably reduce the communications and computation throughput when supporting shared memory.

To address the throughput and overhead issues discussed above, the Intel Paragon machine provides a second processor on each node to act as a messaging processor. The messaging processor is identical to the compute processor, and it resides on a snoopy bus along with the compute processor and the network interface chip. Unfortunately, because Intel's messaging processor is coupled to the compute processor through the snoopy bus, it will be unable to support FLASH's goal of supporting cache coherence. (It needs to be able to observe all requests issued by the compute processor.) MAGIC's tight coupling to the compute processor, the memory system, and the network allows it to support both cache coherence and message passing efficiently.

The recently-announced Meiko CS-2 [HM93] machine incorporates a processor core in its network interface. While the CS-2 is similar to FLASH at a high level, the processor core and the surrounding data path are not powerful enough to be able to implement a cache-coherence protocol efficiently (e.g., there is no on-chip data cache, so all directory data would have to come from main memory). Even messaging is done using a separate DMA controller in the network interface chip. In contrast, in FLASH all messaging and coherence are handled directly by the MAGIC protocol processor.

The *T machine proposed by MIT [NPA92] is in some ways closest to FLASH. *T proposes to use a separate remote-memory processor and a separate synchronization processor in addition to the main compute processor. While the remote-memory and synchronization processors in *T are similar to the MAGIC chip in FLASH, the *T paper discusses them only at a high level, giving the instruction set additions but no hardware blocks. *T also does not have effective support of cache coherence as a goal, and consequently does not discuss the implications of their design for that issue.

In a paper related to *T, Henry and Joerg [HJ92] discuss issues in the design of the network-node interface. They argue that most protocol processing for handling messages can be done by a general-purpose processor. Hardware support is needed in only a couple of places, namely for fast dispatch based on type of incoming message and for boundary-case/error-case checking (for example, the processor should be informed when the outbound network queue is getting full to help avoid deadlock). Our general experience in the design of MAGIC has been similar, although the specifics of our design are quite different.

Finally, the most recent addition to the array of large-scale multiprocessors has been the Cray T3D. The architecture supports a single-address-space memory model and provides a block-transfer engine that can do memory-to-memory copies with scatter/gather capabilities. T3D differs from FLASH in its lack of support for cache coherence and in its support of a more specialized bulk data transfer engine.

8 Concluding Remarks

Recent shared-memory and message-passing architectures have been converging. FLASH is a unified architecture that addresses many of the drawbacks of earlier proposals — it provides support for cache-coherent shared memory with low hardware overhead and for message passing with low software overhead. The goal of supporting multiple protocols efficiently and flexibly has to a large extent driven the architecture of FLASH. To achieve this goal, we made a set of hardware-software trade-offs. For flexibility, MAGIC includes a programmable protocol processor and a programmable hardware dispatch mechanism. In addition, MAGIC uses a portion of the local main memory for storing all protocol code and data instead of employing a special memory for this purpose. For efficiency, MAGIC has hardware support optimized to handle data movement in a high-bandwidth, pipelined fashion, as well as on-chip caches for high-bandwidth, low-latency access to protocol state and code. The result is a single chip that efficiently and flexibly implements all of the functionality required in a scalable multiprocessor system.

Acknowledgments

We would like to thank Todd Mowry for his help in adapting the FlashLite threads package to Tango-Lite and Stephen Herrod for his help with FlashLite development. We would also like to acknowledge the cooperation of the Intel Corporation, Supercomputer Systems Division. This work was supported by ARPA contract N00039-91-C-0138. David Ofelt, Mark Heinrich, and Joel Baxter are supported by National Science Foundation Fellowships. John Heinlein is supported by an Air Force Laboratory Graduate Fellowship. John Chapin is supported by a Fannie and John Hertz Foundation Fellowship.

References

[ACD+91] Anant Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. MIT/LCS Memo TM-454, Massachusetts Institute of Technology, 1991.

[Golds93] Stephen Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[HGG94] John Heinlein, Kourosh Gharachorloo, and Anoop Gupta. Integrating Multiple Communication Paradigms in High Performance Multiprocessors. Technical Report CSL-TR-94-604, Stanford University, February 1994.

[HJ92] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111-22, Boston, MA, October 1992.

[HM93] Mark Homewood and Moray McLaren. Meiko CS-2 Interconnect Elan–Elite Design. In *Proceedings of Hot Interconnects 93*, pages 2.1.1-4, Stanford University, August 1993.

[HP90] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[KA93] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[KSR92] Kendall Square Research. KSR1 Technical Summary. Waltham, MA, 1992.

[LLG+90] Daniel Lenoski et al. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148-59, Seattle, WA, May 1990.

[LLG+92] Daniel Lenoski et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63-79, March 1992.

[NPA92] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156-67, Gold Coast, Australia, May 1992.

[NWD93] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224-35, San Diego, CA, May 1993.

[SGS+93] Ellen Spertus et al. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 302-13, San Diego, CA, May 1993.

[SHL92] Michael D. Smith, Mark Horowitz, and Monica Lam. Efficient Superscalar Performance Through Boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248-59, Boston, MA, October 1992.

[Simoni92] Richard Simoni. Cache Coherence Directories for Scalable Multiprocessors. Ph.D. Thesis, Technical Report CSL-TR-92-550, Stanford University, October 1992.

[Stall93] Richard Stallman. Using and porting GNU CC. Free Software Foundation, Cambridge, MA, June 1993.

[SWG92] J.P. Singh, W.-D. Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5-44, March 1992.

[TMC92] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary. Cambridge MA, January 1992.

[vECG+92] Thorsten von Eicken et al. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256-66, Gold Coast, Australia, May 1992.