The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor

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

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

A flexible communication mechanism is a desirable feature in multiprocessors because it allows support for multiple communication protocols, expands performance monitoring capabilities, and leads to a simpler design and debug process. In the Stanford FLASH multiprocessor, flexibility is obtained by requiring all transactions in a node to pass through a programmable node controller, called MAGIC. In this paper, we evaluate the performance costs of flexibility by comparing the performance of FLASH to that of an idealized hardwired machine on representative parallel applications and a multiprogramming workload. To measure the performance of FLASH, we use a detailed simulator of the FLASH and MAGIC designs, together with the code sequences that implement the cache-coherence protocol. We find that for a range of optimized parallel applications the performance differences between the idealized machine and FLASH are small. For these programs, either the miss rates are small or the latency of the programmable protocol can be hidden behind the memory access time. For applications that incur a large number of remote misses or exhibit substantial hot-spotting, performance is poor for both machines, though the increased remote access latencies or the occupancy of MAGIC lead to lower performance for the flexible design. In most cases, however, FLASH is only 2%-12% slower than the idealized machine.

1 Introduction

The Stanford FLASH (FLexible Architecture for SHared memory) multiprocessor is one of several recent proposals designed to integrate a cache-coherent shared address space and message passing in a single architecture, and to scale cost-effectively from small-scale to large-scale machines [ACD+91, NPA92, NWD93, RLW94]. It achieves these goals by replacing the traditional hardwired node controller with a programmable microcontroller, called MAGIC (Memory And General Interconnect Controller), that can run different code sequences to implement different protocols. All transactions within a node, both those that are generated locally and those that come from remote nodes, go through MAGIC.

To appear in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VI), San Jose, CA, October 1994.

The flexibility of a programmable controller has several advantages: it simplifies the design and debugging process, allows enhancements of functionality, permits experimentation with new protocols, and allows extensive and accurate performance monitoring. Flexibility, however, comes at a cost in performance, since one can always build a hardwired controller that outperforms a flexible one for any given communication protocol.

In this paper, we examine the performance cost of flexibility in the context of supporting cache-coherence on FLASH (the performance of our initial message-passing implementation is discussed in [HGD+94]). We compare the execution time of representative parallel applications and a multiprogramming workload running on FLASH to their run time on an idealized machine in which all node controller operations take zero time.

The flexible node controller in FLASH can cause performance loss for two reasons: (i) the direct contribution that the latency of the node controller makes to memory operations such as cache misses, and (ii) the indirect contribution due to contention resulting from the occupancy of the centralized controller. For many types of transactions, the extra latency of the programmable implementation can be hidden behind the memory access time. Unfortunately, this is not always the case. For example, the latencies of remote operations are significantly higher in FLASH. Our experiments show that these increased latencies usually have a small impact on the performance of optimized workloads.

The occupancy of the flexible node controller is also not a bottleneck for well-written parallel applications. When the occupancy does become large, for example in the presence of significant hot-spotting, we find that it hurts the performance of FLASH relative to the idealized machine only when the controller occupancy is high and the memory occupancy is simultaneously low. The insights from our experiments highlight the potential bottlenecks in scalable cache-coherent machines.

The next section presents a brief overview of the FLASH architecture. A more complete description can be found in [KOH+94]. Section 3 describes our experimental methodology, the machine models we compare, and the applications we use. Section 4 compares the performance of FLASH and the idealized machine. Section 5 evaluates the effectiveness of various architectural features of FLASH. Section 6 presents our main conclusions.

2 FLASH Overview

Every FLASH node contains an off-the-shelf microprocessor, its secondary cache, a portion of the machine's distributed memory, and a flexible node controller called MAGIC, as shown in Figure 2.1. MAGIC implements the interfaces between the node's

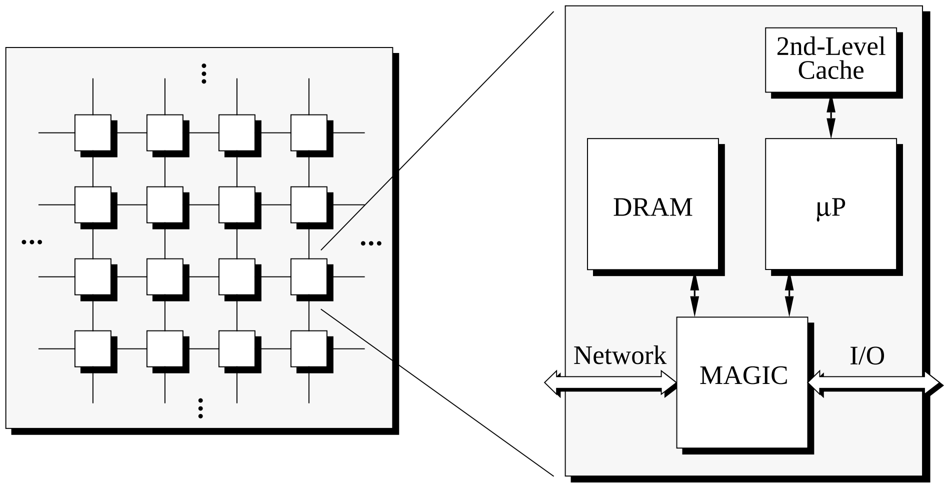

Figure 2.1. FLASH machine and node organization.

processor, local memory system, I/O subsystem, and the interconnection network. As the heart of the FLASH node, MAGIC is responsible for processing all requests from these interfaces and for implementing the cache-coherence and message-passing protocols.

Flexibility demands that MAGIC be able to observe and control all system transactions. MAGIC's central location within the FLASH node satisfies this requirement, but also means that MAGIC can become a performance bottleneck. There are two potential sources of performance loss: the centralized nature of the controller, and the additional processing overhead caused by using a flexible controller rather than a hardwired one. The former source is mostly orthogonal to flexibility. Deadlock avoidance and other issues make the implementation of a decentralized design more difficult, regardless of whether it is flexible. Added latency due to the use of a flexible controller, however, clearly is a performance liability of flexibility. To be conservative, we assume that one could construct a decentralized, hardwired controller and that all performance degradation in FLASH is a cost of flexibility.

MAGIC incorporates several architectural features to make protocol processing efficient, as shown in Figure 2.2. Since protocol processing consists of two largely independent tasks—moving data and updating protocol state—MAGIC separates data movement from control processing, allowing the two tasks to proceed concurrently. The hardwired *data transfer logic* is optimized for low latency and high bandwidth, while the programmable *control macropipeline* handles all protocol processing in a flexible, protocol-independent manner. To see how these paths are used to process a message, we first describe the overall flow of a message through MAGIC (we refer to any inter- or intra-node communication as a *message*). Then we describe the specific architectural features used to enhance efficiency.

When a message enters the MAGIC chip from the processor, network, or I/O subsystem interface (the PI, NI, or I/O, respectively), it is split into message data and a message header. The message data is placed into a *data buffer*, a cache line-sized (128-byte) on-chip storage element used to stage data as it is passed between interfaces. The message header is initially stored in an incoming queue. The first stage in the control macropipeline, the *inbox*, later selects the message header from its queue, preprocesses the header, and passes it to the *protocol processor* (PP). The PP is a general purpose microprocessor core that is responsible for updating protocol data structures, composing outgoing messages, and controlling the data transfer logic. Outgoing message sends are handled by the *outbox*, which accepts outgoing messages from the PP and directs them to the proper destination interface unit. There, the outgoing message headers are combined with their associated data from the data buffer and passed to the processor, network, or I/O subsystem.

To avoid the store-and-forward delays associated with transferring large data blocks, the data transfer logic supports pipelined transfers. Each data buffer is tagged with per-word valid bits and is multiported, allowing a destination interface to read the data while the source interface fills the data buffer. This hardware eliminates the need for multiple data copy operations and makes the latency of a data transfer independent of the transfer size.

In addition to pipelining the control and datapath, MAGIC incorporates several other architectural features to enhance effi-

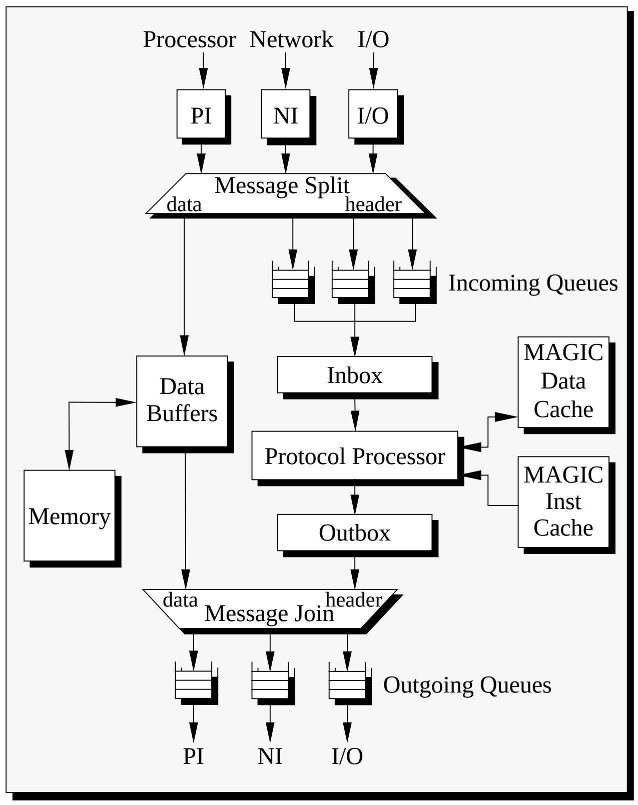

Figure 2.2. MAGIC architecture.

ciency. First, the inbox uses parts of the message header to index into a small associative memory array called the *jump table*. The output of the jump table specifies the starting program counter value for the PP code sequence (or *handler*) appropriate for the message, as well as whether to initiate a speculative memory operation for the address contained in the message header. These inbox-initiated memory operations are called “speculative” because the data retrieved may not actually be needed (e.g., if the requested data is dirty in a remote processor’s cache, the data in memory is not the most up-to-date copy). Issuing the speculative memory operation allows the memory access to begin even before the PP begins processing the message.

Second, to decrease its occupancy, the PP has been optimized for efficient protocol processing. Its instruction set, based on DLX [HP90], has been extended to include bitfield insert/extract and branch on bit set/clear instructions. In addition, the PP is a dual-issue machine, executing a pair of instructions every cycle. To simplify implementation, the PP does not include support for resource conflict detection; all instruction pairs must be statically scheduled to avoid dependencies.

Finally, in FLASH all protocol code and data are maintained in main memory, rather than in separate dedicated storage. To avoid consuming excessive memory bandwidth, the PP accesses this information through the *MAGIC instruction cache* and *MAGIC data cache*, respectively. We explore the individual performance effects of these architectural features in Section 5.

3 Experimental Methodology and Machine Parameters

The results presented in this paper are gathered from FlashLite, the FLASH system-level simulator. FlashLite is a multithreaded memory system simulator that interfaces to the Tango Lite [Golds93] event-driven reference generator and to the SimOS environment [RV94]. It accurately models the latencies and contention effects in the system, using cycle counts and arbitration information from our working Verilog model of MAGIC. The availability of accurate timing information is a crucial aspect of our performance evaluation, since the performance impact of flexibility depends heavily on the cycle counts one assumes.

3.1 Machine Models

We simulate the MAGIC design we plan to use in the real machine, including detailed parameters such as queue depths, number of data buffers, and the time to retrieve data from the processor’s cache. Table 3.1 summarizes the queue and buffer sizes in MAGIC, and the consequences of exceeding them. As noted earlier, we assume an idealized implementation for the hypothetical hardwired machine (which we refer to as the *ideal machine*). Specifically, we replace MAGIC’s macropipeline with an idealized controller that can process all protocol operations in zero time. The only delays that the ideal machine encounters are those due to contention for shared resources (such as the processor bus, memory system, and network) and data transfer delays. We further assume an infinite depth for all network and memory system queues, which means that the ideal machine will never stall waiting for queue space to become available. The ideal machine uses the same cache-coherence protocol as FLASH (see Section 3.3). However, because it processes all protocol operations in zero time, its directory implementation is irrelevant and can be thought of as an instantaneous oracle.

Of course, this ideal machine cannot actually be built. Protocol operations would take some time even in a hardwired implementation. Moreover, it is unclear that a deadlock-free implementation can be built without any central serialization of events within a node. However, the protocol overhead and amount of required serialization in a real implementation is subject to debate, and using this idealized model gives us an upper bound on the performance costs of MAGIC’s flexibility.

3.2 Common Characteristics

Both FLASH and the ideal machine employ the same compute processor, cache, network, and memory system. We assume an aggressive, 400 MIPS compute processor as being representative of state-of-the-art next-generation microprocessors. Since faster processors issue more memory requests per unit time, our choice of a very fast processor exacerbates the performance costs of FLASH’s flexibility. MAGIC operates at its target speed of 100 MHz. Thus, the processor can issue up to four memory requests (peak) during each system clock cycle. All cycle counts we report in the paper are expressed in 10 ns system clock cycles.

The processor is assumed to have blocking reads but non-blocking writes: a write will stall execution only if another miss is outstanding to a line that maps to the same cache index as, but has a different tag than, the current request. A write that maps to the same cache index and has the same tag as an outstanding miss will be merged with that miss and will not stall the processor. The processor cache is two-way set associative, has a line size of 128 bytes, and supports up to 4 outstanding cache misses. In addition, to minimize its miss penalty, the cache assumes critical-word-first data return. The processor implements its own cache control, so MAGIC must issue a processor bus transaction to retrieve data from or perform invalidations to the processor’s cache.

Both machines use a 128-byte cache line and a 64-bit path to the memory system. We assume a 14-cycle memory access time for both machines, a figure that is somewhat smaller than the actual memory system latencies of current uniprocessor worksta-

Table 3.1. MAGIC Resource Limits

Resource	Size	Impact if full/unavailable
Incoming network queues	16 messages	Messages back up into the network
Outgoing network queues	16 messages	PP stalls until outgoing queue space is available
Memory controller queue	1 request	PP or inbox stalls until queue entry is available
Inbox-to-PP queue	1 message	Inbox stalls until PP reads new message from queue
Outgoing PI queue	1 message	PP stalls on next send until PI can accept a new message
Incoming PI queue	16 messages	Processor stalls until queue space is available
Data buffers	16 buffers	Unit needing a data buffer stalls until one is available

tions.[1] The access time refers to the number of cycles from the time a memory operation reaches the front of the memory controller's queue to the time the first 64 bits of data are returned to the node controller. FlashLite models memory system contention accurately.

Any time a message enters the network, it is charged a fixed network transit latency. This latency is based on the average transit time for a two-dimensional mesh network having a per-hop fall-through time of 40 ns [Intel94]. For our 16-processor simulations, the average message requires latency equivalent to one hop to both enter and exit the network, 2.6 hops of network transit, and 3 cycles of network header information, yielding an average transit time of 220 ns, or 22 cycles.

3.3 Memory Latencies and Occupancies

In Table 3.2, we list the suboperations of a memory request and compare the latencies for the actual MAGIC chip to those assumed for the ideal machine. For the ideal machine, the only latencies are those representing data transit or arbitration; all other macropipeline suboperations are assumed to complete in zero time.

The "varies" entry in the table for PP processing time in FLASH indicates that the time required to service a request varies with the type of request, since different requests invoke different code sequences on the PP. The time to service a request also depends on the directory organization. The initial protocol we will run on the FLASH prototype is the *dynamic pointer allocation* protocol [Simoni92, KOH+94]. In this scheme, each main memory line has an associated *directory header* which contains some status bits and a link to a linked list of sharing nodes. Most protocol operations require access to the directory header. Depending on the state of the line and the type of request, a traversal of the linked list of sharers may also be needed.

For accuracy in our performance evaluation of flexibility, we took the hardware suboperation latencies from the MAGIC Verilog model and the protocol code latencies from an instruction set emulator we wrote for the PP and integrated with FlashLite. The protocol handlers that we plan to run on the real PP have been coded in C and compiled using a port of the *gcc* compiler [Stall93] to the PP architecture. The compiler output is scheduled using PPtwine, a port of the superscalar instruction scheduler originally designed for the Torch project [Smith92]. Finally, the output of PPtwine is hand-tuned slightly to overcome some limitations in the current software tools. PPsim, the instruction set emulator for the PP, executes the handlers and reports accurate instruction usage statistics and dynamic cycle counts.

Table 3.3 lists the latencies of common memory operations (cache read misses) in FLASH and the ideal machine, assuming no contention. The latency numbers in the table represent the total number of cycles required to service the request from the time the processor detects the miss in its cache to the time the first 8 bytes of data are returned on the processor bus. In addition, for each type of read miss, Table 3.3 lists the total PP occupancy of *all* the handlers that must be run to satisfy that miss. These occupancies are further broken down into their individual components in Table 3.4.

To make the derivation of the latencies in Table 3.3 more concrete, Figure 3.1 depicts the suboperations of a local read miss to a clean line for both FLASH and the ideal machine. Note how the MAGIC PP handler time is overlapped almost completely with the local memory access. To the extent that the protocol processing can be hidden in this manner, the latency of the flexible processing does not impact the total request service latency, and the only possible drawback is the increased latency due to the occupancy of the PP. Unfortunately, the potential to overlap PP processing with memory access is reduced for remote memory operations. Since a remote message requires processing by at least two nodes, the overhead of entering and exiting the nodes leads to a greater discrepancy in the remote memory operation latencies than in the local memory operation latencies.

In addition to latencies incurred when servicing memory references, other protocol transactions also impact PP occupancy in FLASH. Specifically, message types such as invalidation requests, invalidation acknowledgments, and replacement hints do not require memory system access, but servicing these messages can require a significant number of PP processing cycles. The added

Table 3.2. Suboperation Latencies in 10 ns Cycles

Suboperation	MAGIC Latency	Ideal Latency
Processor:		
Miss detect to request on bus	5	5
Bus transit	1	1
Processor Interface:		
Inbound processing	1	1
Outbound processing	4	2
Outbound bus arbitration	1	1
Outbound bus transit for 1st word	1	1
Retrieve state from processor cache	15	15
Retrieve first double word of data from processor cache	20	20
Network Interface:		
Inbound processing	8	8
Outbound processing	4	4
Inbox:		
Queue selection and arbitration	1	1
Jump table lookup	2	N/A
Protocol Processor:		
Handler execution	Varies	N/A
MDC miss penalty	29	N/A
Outbox outbound processing	1	N/A
Network transit, average case	22	22
Memory access, time to first 8 bytes	14	14

Table 3.3. Memory Latencies and Occupancies, No Contention, in 10 ns Cycles

Operation	Ideal Latency	FLASH Latency	FLASH PP Occ.
Local read miss:			
Clean in local node's memory	24	27	11
Dirty in remote cache	100	143	53
Remote read miss:			
Clean in home node's memory	92	111	16
Dirty in home node's cache	100	145	53
Dirty in a 3rd node's cache	136	191	61

1. The DEC Alpha 3000, for example, performs a 32-byte cache fill in 180 ns [DEK+92]. The current SPARCStation 10 machines require 190 ns to the first word; new models will reduce this time to 160 ns.

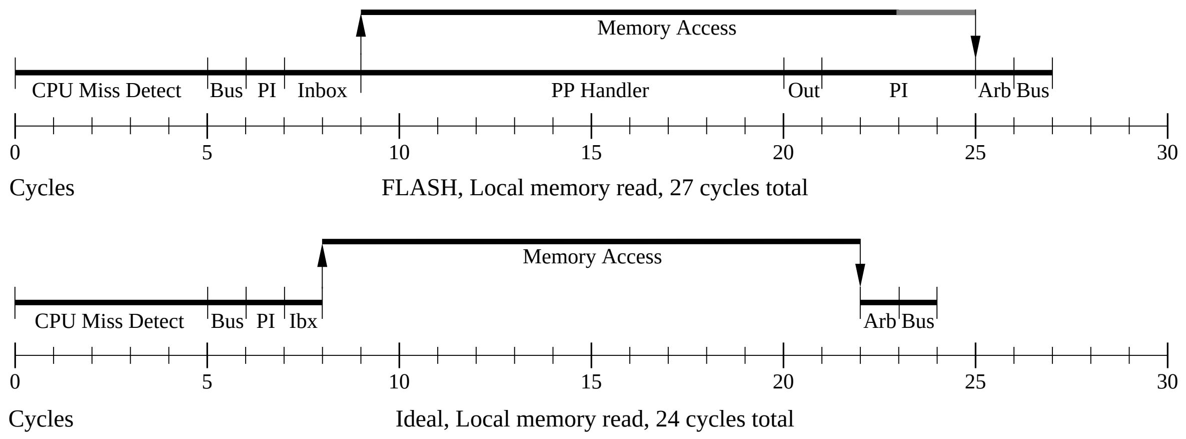

Figure 3.1. Suboperations of a local memory read.

PP occupancy leads to additional queuing delays, which can increase the latency of subsequent memory operations. In comparison, the ideal machine assumes zero PP occupancy and can service these messages continuously and infinitely fast, avoiding the queuing delays encountered in FLASH. Thus, FLASH has both latency and occupancy disadvantages relative to the ideal machine.

Table 3.4. PP Occupancies for Common Operations, in 10 ns Cycles

Operation	PP Occupancy
Service read miss from main memory	11
Service write miss from main memory	14 + 10 to 15 per invalidation
Forward request to home node	3
Forward request from home to dirty node	18
Retrieve data from processor cache	38
Forward reply from network to processor	2
Local writeback	10
Local replacement hint	7
Writeback from a remote processor	8
Replacement hint from remote processor:	
Processor is only node on sharer list	17
Processor is Nth node on sharer list	23 + 14 * N

3.4 The Applications

We used two main criteria in choosing the applications for our evaluation: they should be representative of important classes of computations likely to be run on a multiprocessor like FLASH, and they should span a range of memory referencing and communication patterns. Since we intend FLASH to operate in both traditional supercomputing environments and general-purpose multiprogramming environments, we focused on two categories of applications: parallel scientific applications, and OS multiprogramming workloads. The applications, the computational classes they represent, and the input data set sizes we use for them are listed in Table 3.5. Descriptions of the parallel applications can be found in: LU and FFT [RSG93]; Ocean and Radix [WSH94]; Barnes and MP3D [SWG92].

The problem sizes we use for the parallel applications are realistic, and would be run on 16-processor machines in practice. However, owing to the costs of simulation, they clearly are not the largest problem sizes one would run. Some of the applications have two important operating points, both of which are representative on modern machines such as FLASH: one in which the important working set of a processor fits in its cache, and the other in which it does not (for example, in a regular-grid iterative computation, a processor's partition may or may not fit in the cache, depending on the size of the grid being used). Simulating the 1 MB caches in FLASH with the problem sizes we use, we are able to capture only the first of these modes, so that we see only the effects of inherent communication, false sharing, and cold misses, but not of misses due to cache capacity.

To capture the behavior in cases where the working set does not fit in the cache, we ran the same applications with two smaller cache sizes: 4 KB and 64 KB. The 4 KB size is smaller than the working set in all cases, and the intermediate 64 KB size is useful because working sets are sometimes not clearly defined with low-associativity caches [RSG93]. For some applications, the smaller cache simulations either are not useful, or are not representative of any realistic situation that would occur with 1 MB caches on the real machine. As a result, we do not simulate LU or the OS workload with either 64 KB or 4 KB caches, and we do not simulate Barnes with 4 KB caches.

To evaluate the performance of the machines under an OS workload, we use the SimOS environment to capture all code and data references from both the operating system and user-level code, and model the hardware of a multiprocessor in enough detail to boot an operating system and run a workload. The multiprogramming simulations are parameterized with the same CPU and cache model as the parallel application simulations and use FlashLite as the memory system simulator. To remove I/O delays and focus the comparison on CPU and memory system performance, the simulation environment models a zero-latency system disk that can DMA into any node's memory.

The OS workload uses a port of Silicon Graphics IRIX 5.2, an SVR4 Unix-based operating system. IRIX is a symmetric multiprocessing kernel with relatively fine-grained locking that has been optimized for bus-based machines with uniform memory access times. Since larger configurations of our NUMA machines spend more than half their execution time waiting on locks in IRIX, we use only 8-processor configurations for the OS workload. Because IRIX lacks a NUMA memory allocator, we allocate the physical pages of the machine round-robin across the local node memories. On top of IRIX, we ran eight copies of a "make" of a small pro-

Table 3.5. Applications and Problem Sizes

Application	Representative Of	Problem Size
Barnes	Hierarchical N-body codes	8192 particles, θ=1.0
FFT	Transform methods, high-radix	64K complex points, radix $\sqrt{N}$
LU	Blocked dense linear algebra	512-by-512 matrix, 16-by-16 blocks
MP3D	High-communication unstructured accesses	50,000 particles
Ocean	Regular-grid iterative codes	258-by-258 grids, 25 grids
OS	Multiprogramming environments	8 "makes" of a 2809-line C program
Radix	High-performance parallel sorting	256K integer keys, radix=256

gram containing two source files. The "make" runs the passes of the C compiler on the sources and links the object files into an executable. This workload was chosen because of its high OS activity—close to 50% of the time is spent in kernel mode. The workload is particularly stressful for the operating system, making heavy use of the file system, virtual memory, and process management subsystems.

4 FLASH versus the Ideal Machine

We now present the results of our simulations for the seven applications described above. In Section 4.1 we discuss the results with 1 MB caches and then, to explore the effects of working sets larger than the cache size, we show results for 64 KB and 4 KB cache simulations in Section 4.2. The parallel application experiments are run for 16 processor systems, and the OS workload is run for 8 processor systems. Initial 64-processor parallel application results are discussed in Section 4.5.

4.1 Results with 1 MB Caches

Figure 4.1 shows the execution time difference between FLASH and the ideal machine for each application with 1 MB caches. The number at the top of each bar indicates the total application execution time, normalized so that the FLASH execution time is 100. The smaller numbers alongside the execution time bars indicate the height of the bar for that component of the execution time. Execution time is allocated to five categories: processor busy time (Busy), contention for the cache (Cont), read stall time (Read), write stall time (Write), and time spent waiting for synchronization (Sync).

Table 4.1 summarizes the processor cache miss rates, read miss distributions, and average main memory and PP occupancies for these experiments. The contentionless read miss time (CRMT) is calculated by multiplying the latencies in Table 3.3 by the distribution in Table 4.1. With the exception of MP3D, the miss rates for all of the parallel applications were very low (less than 1%). While these miss rates are small, they are representative of optimized parallel applications that make efficient use of FLASH's 128-byte cache lines, and whose working sets fit in the cache. Because the applications generate few cache misses, the occupancy of the protocol processor does not impact performance. In fact, the processor utilization is high enough that the 35% average difference in CRMT results in only 2% to 10% difference in overall execution time. MP3D, which is our communication stress test, has a much higher miss rate and spends most of its time in the memory system. Particularly because most of the misses are dirty in a remote cache, the execution time of MP3D is 25% larger on FLASH than on the ideal machine. However, communication is so high that even the ideal machine performs poorly on this application.

For the six parallel applications, the performance difference can be derived analytically by taking the ratio of FLASH CRMT to ideal CRMT and multiplying it by the fraction of time stalled in the memory system for the ideal machine. Thus, for these experiments, where the working sets fit in the cache, the performance difference between the FLASH and ideal machines is governed solely by their inherent latency difference.

The results for the multiprogramming workload cannot be explained by latency differences alone. The occupancy of the protocol processor is quite high for this workload, reaching a maximum of 39%. Some of the occupancy increase is caused by MAGIC Data Cache misses, which we discuss in Section 5.2, and the rest by mild hot-spotting in the memory system. Hot-spotting tends to favor the ideal machine in this study, since it results in queueing delays in the FLASH machine that are not present in the zero-occupancy ideal machine. We examine hot-spotting and occupancy effects further in Section 4.3.

4.2 Results with Smaller Caches

Let us now examine the cases where the working sets do not fit in the processor cache by using smaller caches as discussed in Section 3.4. The introduction of substantial capacity misses not only increases the number of misses that must be processed, but in some cases also increases the communication to computation ratio. The cache miss rates and their distributions with the intermediate (64 KB) and small (4 KB) caches are shown in Table 4.2.

Figure 4.2 and Figure 4.3 depict the execution time breakdowns for the 64 KB and 4 KB cache runs, respectively. As expected, both machines show a significant drop in the processor utilization and a significant increase in the cache miss rate. Because the applications are now spending a larger fraction of their execution time stalled on the memory system, one might expect that the FLASH machine's performance would degrade further from the ideal machine's performance. However, this is not necessarily the case. For example, the relative performance of the FLASH machine for Radix improves with 64KB caches, while that for FFT, Ocean and MP3D remains about the same. The reason is that the distribution of misses now is dramatically different than the distribution when the working sets fit in the cache. As shown in Table 4.2, in most cases many more misses are satisfied locally, a case for which the latency difference between FLASH and the ideal machine is small. Applications that require high local memory bandwidth thus perform only marginally worse on FLASH than on the ideal machine.

In most cases, the performance differences between FLASH and the ideal machine can still be attributed entirely to differences in latency, by using the CRMT-based calculation in Section 4.1.

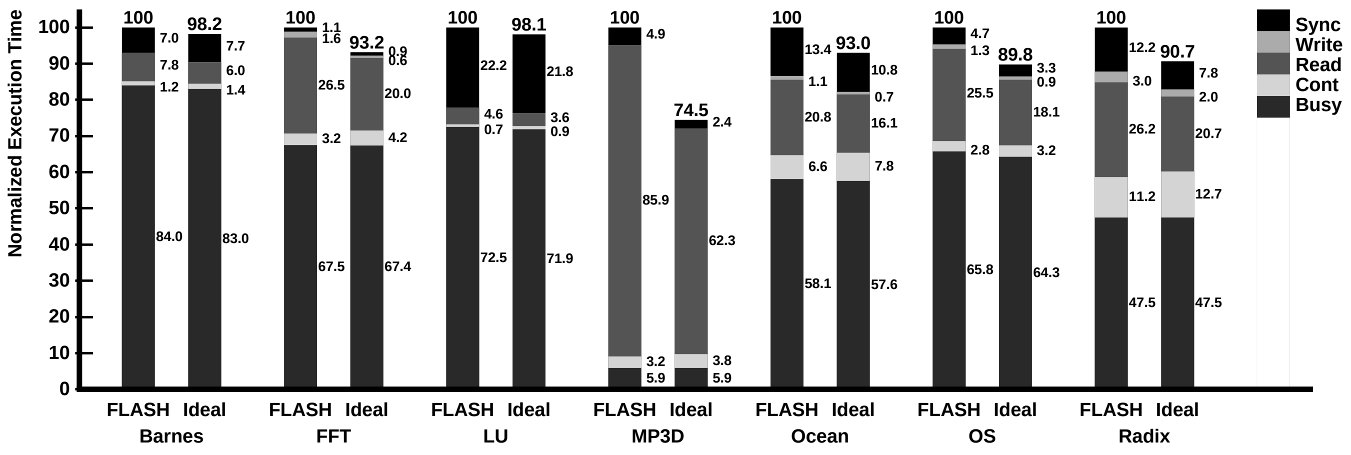

Figure 4.1. Execution times for FLASH and the ideal machine, 1 MB caches.

Table 4.1. Read Miss Distributions and Contentionless Read Miss Time (CRMT) in 10 ns Cycles, 1 MB Caches

	Barnes	FFT	LU	MP3D	Ocean	OS	Radix
Miss Rate	0.06%	0.64%	0.05%	6.00%	0.91%	0.09%[a]	0.78%
Local Clean	2.4%	20.1%	1.0%	0.4%	51.7%	20.0%	2.6%
Local Dirty Remote	3.7%	0.0%	0.0%	5.9%	0.0%	2.7%	76.0%
Remote Clean	38.7%	17.7%	67.1%	3.8%	10.5%	58.6%	16.6%
Remote Dirty at Home	3.6%	62.1%	31.9%	5.9%	37.8%	2.6%	2.2%
Remote Dirty Remote	52.6%	0.1%	0.0%	84.0%	0.0%	16.1%	2.6%
FLASH CRMT	153	115	121	182	80	109	136
Ideal CRMT	114	83	94	130	60	86	98
Avg. Mem Occupancy	4.2%	8.2%	0.8%	7.0%	13.0%	9.9%	8.7%
Avg. PP Occupancy	5.4%	14.3%	1.7%	36.2%	17.7%	21.0%	22.8%

a. Unlike the other simulations, the OS workload includes the effects of instruction cache misses from both the application and the operating system. The instruction cache miss rate was 0.025% and the data cache miss rate was 0.325%.

Table 4.2. Read Miss Distributions and Contentionless Read Miss Times (CRMT) in 10 ns Cycles, Smaller Caches

	Barnes	FFT		MP3D		Ocean		Radix	
	64 KB	4 KB	64 KB	4 KB	64 KB	16 KB	64 KB	4 KB	64 KB
Miss Rate	0.6%	8.7%	1.1%	7.5%	7.1%	11.4%	2.5%	10.0%	4.2%
Local Clean	7.0%	64.7%	42.7%	3.8%	1.4%	95.6%	88.6%	91.3%	80.1%
Local Dirty Remote	0.1%	0.0%	0.0%	2.8%	4.7%	0.0%	0.0%	0.0%	5.9%
Remote Clean	91.1%	35.3%	45.1%	50.2%	20.6%	4.0%	7.3%	8.2%	11.9%
Remote Dirty at Home	0.1%	0.0%	12.2%	2.8%	4.7%	0.4%	4.1%	0.1%	0.8%
Remote Dirty Remote	1.7%	0.0%	0.0%	40.4%	68.6%	0.0%	0.0%	0.4%	1.3%
FLASH CRMT	107	57	79	142	168	31	38	35	47
Ideal CRMT	88	48	64	108	122	27	32	30	39
Avg. Memory Occupancy	9.4%	32.6%	10.6%	8.8%	7.6%	28.0%	20.7%	33.5%	29.0%
Avg. PP Occupancy	23.0%	36.5%	15.2%	32.0%	35.6%	29.8%	22.1%	35.1%	30.6%

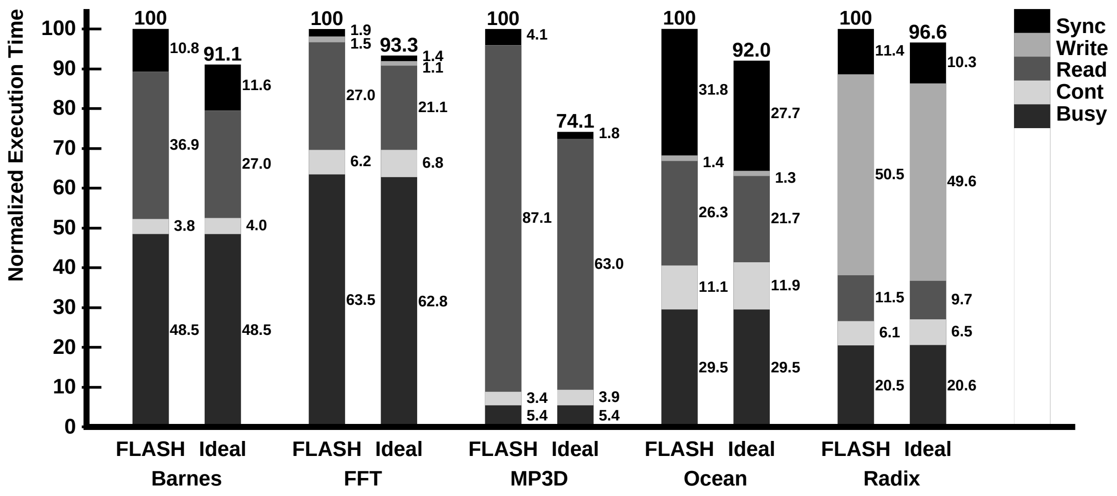

Figure 4.2. Execution times for FLASH and the ideal machine, 64 KB caches.

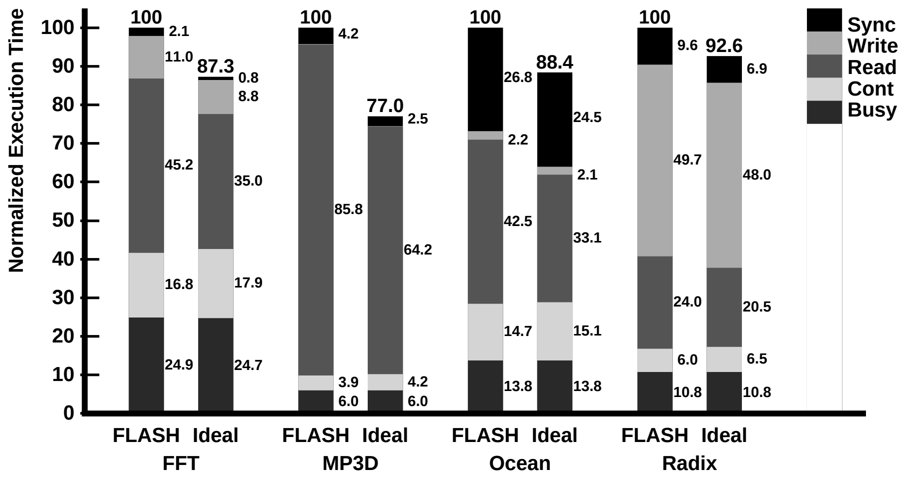

Figure 4.3. Execution times for FLASH and the ideal machine, 4 KB caches (16 KB for Ocean)

For Barnes and Ocean[2] at the smallest cache size, however, PP occupancy begins to impact performance as well. While in these cases the occupancy-induced performance loss is small (less than 2%), there are other situations in which PP occupancy can have a more substantial effect, as described in the next section.

4.3 Effects of PP Occupancy

Applications which are less optimized can have higher PP occupancies than we have seen so far, particularly in the presence of hot-spotting. However, high PP occupancy in itself does not cause poor FLASH performance relative to the ideal machine. Poor relative performance requires the simultaneous occurrence of high PP occupancy and low memory occupancy at a node. When memory occupancy is high, the latency of the protocol processor can be hidden behind the memory access, and PP occupancy does not substantially impact performance. To demonstrate this, we ran a version of FFT with 4 KB caches that allocated all of its memory from node zero. The PP occupancy of that node was 81.6%, but the difference in execution time between FLASH and the ideal machine was only 2.6% because the memory occupancy of node zero was also high at 67.7%.

An application in which protocol processor occupancy is much greater than the main memory occupancy is the original port of IRIX we used for our OS workload. This version corresponds to taking IRIX as written for a bus-based machine and running it on a NUMA machine. In particular, it does not allocate pages round-robin among all the memories on the machine, but rather fills the memory of one node before going on to the next node. The maximum PP occupancy in this case was 81% while the maximum memory occupancy was only 33%. This led to a 29% performance degradation for FLASH relative to the ideal machine. Similar results are observed in less optimized versions of some parallel applications, which generate substantial hot-spotting on data that are dirty in a node's cache.

Note that the ideal machine is particularly optimistic with respect to occupancy. For example, to issue five invalidations the PP would be occupied for 5*15 or 75 cycles, while the ideal

2. Because of cache conflict problems with a 4 KB cache at 128-byte lines, we used a 16 KB cache for our Ocean simulations.

machine would have no occupancy. A real hardwired implementation would also incur occupancy-related contention, so the degradation we observe relative to the ideal machine is simply an upper bound. Finally, the results for our workloads show that the occupancy of the PP is comparable to the read miss stall time, indicating that it would be difficult to perform protocol processing on the main processor without adversely affecting performance.

4.4 Summary of Results

Overall, we find that for optimized parallel applications the performance impact of flexibility in FLASH varies between 2% and 12% over the ideal machine for a range of important computations. In most cases, the performance discrepancies can be traced entirely to differences in latency between the two machines, and PP occupancy does not have a substantial effect. Thus, the performance difference is larger for applications that have many remote misses, and generally smaller when most misses are local and the protocol processing latency can be hidden under the memory access time. The differences become larger when PP occupancy becomes a bottleneck, which we have observed for less tuned applications that exhibit substantial hot-spotting. Finally, we note that while the flexibility of the PP can lead to occupancy-related problems due to hot-spotting, the same flexibility can be used to dynamically detect hot-spotting situations and provide support for techniques such as automatic page remapping or migration.

4.5 Scaling to Larger Machines

We have performed some initial experiments with 64-processor configurations for the parallel applications. To stress communication in these initial experiments, we used the same problem sizes as in the 16-processor runs, so the problem sizes were significantly smaller than might be considered realistic for 64-processor systems. The use of smaller problems drives up the communication to computation ratio and the remote miss rates, and thus reduces the relative performance of FLASH (for example, performance differences were 17% for FFT and 12% for Ocean, although the difference for LU remained small at only 0.7%). How the relative performance would scale in practice depends on how the problem sizes are scaled—the more the problem size is scaled, the smaller the relative performance difference between FLASH and the ideal machine. For example, scaling the data set size proportionally for the FFT application resulted in only a 12% performance degradation.

5 MAGIC Architectural Evaluation

As noted in Section 2, MAGIC contains a number of architectural features to accelerate protocol processing. In this section we study the effectiveness of three of these: the speculative memory initiation provision in the inbox, the performance of the MAGIC caches, and the PP architecture extensions.

5.1 Speculative Memory Initiation

To achieve low latency for local memory operations, the memory access must be started as quickly as possible. Therefore, for some types of messages, the inbox jump table indicates that a speculative memory operation should be issued even before the PP processes the message. Unfortunately, some of these speculative operations will be *useless* because the most up-to-date copy of the data may be held in a processor's cache rather than in the node's local memory. Although a useless memory operation does not delay processing of the associated message, it does occupy the memory system for the duration of the access, potentially delaying the initiation of subsequent memory accesses.

To assess the performance impact of useless speculative reads, we ran the set of six parallel applications and our multiprogramming workload with the jump table programmed normally and then with all speculative memory operations disabled. In the latter case, the PP is responsible for initiating the memory access after reading the directory state, and there are no useless memory reads. Table 5.1 shows the fraction of useless speculative reads as well as the overall performance degradation without speculation for each application. Speculation is always beneficial, indicating that the delay in issuing the read outweighs any potential disadvantage of useless speculative reads unnecessarily loading the memory system. A second, subtler, point is that the length of the handlers that cause speculative reads are always about equal to or longer than the memory access time. Consequently, even if a speculative memory access is issued in error, it will finish by the time the next handler runs. With small caches the performance advantages of speculation increase because more cache misses are satisfied locally, and a larger portion of the application's execution time is spent in the memory system. Thus applications that require large local memory bandwidth benefit substantially from speculative reads.

Table 5.1. Impact of Speculative Memory Operations

	1 MB Cache		4 KB Cache[a]	
Application	**Useless reads, w/ speculation**	**Execution time increase, w/o speculation**	**Useless reads, w/ speculation**	**Execution time increase, w/o speculation**
Barnes	54.0%	12.7%	N/A	N/A
FFT	43.5%	0.9%	5.9%	6.8%
LU	33.5%	0.2%	N/A	N/A
MP3D	67.8%	11.8%	37.7%	11.4%
Ocean	20.0%	2.2%	1.2%	21.0%
OS	21.9%	2.9%	N/A	N/A
Radix	59.9%	4.8%	18.0%	17.9%

a. As mentioned in Section 4.2, we use a 16 KB cache for ocean instead of a 4 KB cache.

5.2 MAGIC Data Cache

As mentioned earlier, the protocol processor uses the on-chip MAGIC data cache (MDC) to reduce the average cost of accessing protocol data structures from main memory. Clearly the performance of the overall machine depends on the hit rate in this cache.

The MDC is 64 KB in size, 2-way set associative, and has 128-byte lines. Since in our initial protocol each directory header (see Section 3.3) is eight bytes, each MDC line can hold 16 directory headers, each of which contains the directory information for one 128-byte memory line. Thus, each MDC line contains the directory headers for 16 * 128 = 2 KB of contiguous data. The entire MDC has 512 lines, so the MDC as a whole can hold the directory information for up to 1 MB of data. With this size cache, the MDC miss rates for our parallel application set were too small to affect performance significantly (0.84% overall miss rate, 1.43% read miss rate).

To better understand the effect MDC misses could have on the overall machine, we looked for realistic applications that would stress the MDC. Finding these applications was difficult because such a parallel application must have both a pattern of cache misses that stress the MDC, and a very large data set, since the MDC at a node contains protocol information only for data that are allocated in that node's local memory, and therefore about 1/P of the total data set. An application that streams through a per-processor data set of more than 1 MB with unit stride is not sufficient to stress the MDC. This application will miss in the processor cache in 1 out of every 16 processor references (128-byte lines store 16 8-byte values), and the MDC will miss 1 out of every 16 processor cache misses (since an MDC line maps 16 contiguous processor cache lines). But adding one MDC cache miss penalty to every 256 processor references will have only a small effect on total execution time.

A more pathological situation with regular access patterns is one in which a processor traverses a large local data set with stride greater than 2 KB. However, few well-written codes for a non-vector machine will do this. A naive matrix transpose would, but in reality a matrix transpose would be blocked to get reuse of long cache lines in the processor cache. Even if the transpose is not blocked, it will require the transpose of a matrix with at least 512 rows *per processor* (in the two-dimensional case) to generate a significant number of MDC misses.

Applications with irregular access patterns also must be considered. Sparse matrix computations are an important example. However, direct sparse matrix solvers would also be blocked and would not cause many MDC misses. Iterative sparse problems are essentially matrix-vector product computations in which the matrix is traversed with unit stride, and the vector size per processor is too small to trouble the MDC.

One interesting application that can potentially stress the MDC is the radix sort program. When the radix is large enough (greater than the number of lines in the MDC), the application generates fairly random write references with large enough stride to use the MDC very inefficiently. On a parallel machine, the array being written is distributed among the processors, so that it takes a large problem size to stress the MDC on a machine with more than a few processors. To evaluate the ability of radix sort to stress the MDC, we increased the data set size to 16 MB, used a radix of 2048, and used only one processor. Although the processor cache miss rate was 1.4%, the overall MDC miss rate was 14.9%, with a 30% MDC read miss rate. (The MDC write miss rate is approximately zero since almost all directory operations involve a read-modify-write on the directory state.) As a result, FLASH ran 14% slower than a uniprocessor with no MDC miss penalty.

Another application that can stress the MDC is the operating system. Our OS workload had an overall MDC miss rate of 4.1%, and an MDC read miss rate of 8.7%. One node's PP had an MDC read miss rate as high as 13%. Writebacks and replacement hints cause most of the MDC misses because conflicting lines in the processor cache can also conflict in the smaller MDC. MDC misses and their associated writebacks accounted for 34% of the total memory operations, and were responsible for 2.7% of the 10% difference in execution time between FLASH and the ideal machine shown in Figure 4.1. These MDC misses increase PP occupancy and hence the latencies of subsequent memory requests.

Although the MDC can affect performance in some situations, our results and the above arguments show that it is not likely to be a performance bottleneck for most applications, particularly on moderate to large-scale machines. Problems can arise when large data sets are run on small-scale machines—but even then only for applications with certain types of irregular, high-stride access patterns—or when protocol operations generate a lot of MDC misses. In these cases, one can always exploit the flexibility of MAGIC to implement a coherency protocol that uses the MDC more efficiently.

5.3 PP Architecture Extensions

To speed up common coherence protocol operations, we have extended the PP architecture beyond that of a standard embedded RISC CPU core by adding bitfield insert/extract and branch on bit set/clear instructions, and by making the PP a statically scheduled, dual-issue processor. In Table 5.2 we summarize some statistics which allow us to gauge the utility of the PP architecture extensions. The data in this table are based on our implementation of the full cache-coherence protocol, including all corner cases, deadlock avoidance checks, and other complications. As the table shows, the total size of the cache-coherence code sequences is about 15 KB,

Table 5.2. PP Architecture Evaluation

Parameter	1 MB Caches	64 KB Caches	4 KB Caches
Static code size of fully-scheduled handlers (with NOPs)	14.8 KB		
Dynamic dual-issue efficiency[a]	1.53	1.54	1.43
Special instruction use (dynamic fraction of ALU and branch instructions that are bitfield or branch-on-bit)	38%	37%	43%
Mean number of instruction pairs executed per handler invocation	13.5	13.1	10.8
Mean number of handler invocations per processor cache miss	3.69	3.87	3.51

a. Dynamic dual-issue efficiency is the ratio of the dynamic count of non-NOP instructions executed to the total number of dual-issue instruction pairs executed. Perfect use of the PP's dual-issue capability would yield a dynamic dual-issue efficiency of 2.

Table 5.3. Comparison of Special Instructions to DLX

Instr Type	DLX Substitution Code Static Size	DLX Substitution Code Latency
Find first set bit	6 instructions (optimized for code size) 27 instructions (optimized for speed)	2 cycles + 4 cycles per bit checked 7-21 cycles (depending on bit position)
Branch on bit	2 or 4 instructions (depending on bit position)	2 or 4 cycles
ALU field immediate	1-5 instructions OR 2 instructions if the immediate is loaded from the data section	1-5 cycles OR 3 cycles (load, load delay, op) plus average cache miss penalty
Insert field	Equivalent to two field immediates (as above) followed by an “or”	

well below the MAGIC instruction cache size of 32 KB. As a result, the only MAGIC instruction cache misses in our simulations are cold misses. In addition, the dynamic dual-issue efficiency and special instruction use are both high, indicating that the typical code sequences make good use of the PP extensions.

The special instructions fall into four general categories: find first set bit; branch on bit set or clear; general ALU field immediate instructions, which specify an immediate operand as a string of consecutive ones or zeros; and field insertion, which overwrites a field in the target with the corresponding field from the source. Table 5.3 shows the number of DLX instructions that would be required to replace each type of special instruction as well as the time required for the DLX substitution code (as opposed to one cycle for each special instruction, not counting branch delay slots). In general, the substitution code consists of sequentially dependent instructions.

To quantify the effect that the extensions have on overall performance, we modified our compiler so that it generated code that did not use any of the special instructions. We scheduled that code for a single-issue PP, and ran our six parallel applications using that version of the protocol. The average performance degradation with the non-optimized PP was found to be 40%, and the maximum performance degradation was 137% (for MP3D). Clearly, an optimized PP is essential to minimizing the performance cost of flexibility.

6 Summary and Conclusions

A flexible node controller offers many advantages: easier debugging, support for a variety of protocols, performance monitoring, and simpler design. To obtain its flexibility, the Stanford FLASH Multiprocessor employs MAGIC, a flexible node controller with a general-purpose processor core. But flexibility comes at a cost in performance relative to a hardwired implementation. The performance loss stems from two sources: the latency that the flexible node controller adds to memory operations, and the contention resulting from the increased occupancy of the node controller itself. We have examined the performance costs of flexibility in FLASH by comparing it to an idealized hardwired implementation in which protocol operations take zero time to process.

For a range of important, optimized parallel applications, the performance loss can be attributed almost entirely to the additional latency of the flexible node controller. The latency disadvantage for FLASH is smaller for local misses, where the overhead of the flexible controller can be hidden behind the memory access time, and larger for remote misses where it cannot. Thus, the performance loss is higher for those applications with high communication to computation ratios, and lower for those applications whose main requirement is local memory bandwidth. Overall, we find that the performance impact of flexibility is between 2% -12% for a range of important computations, including an operating system and multiprogramming workload.

The occupancy of the flexible controller is not a performance liability for our optimized parallel applications, but it can become a problem for less optimized applications that exhibit significant hot-spotting. However, we find that the occupancy of the controller is a performance bottleneck only when it is high and the main memory occupancy on the node is simultaneously low. Our occupancy results (even for the optimized parallel applications) indicate that using the main processor to process protocol operations may result in significant performance loss, unless the additional main processor occupancy can be hidden under the memory access time.

While our results indicate that a machine with a flexible node controller can have comparable performance to a machine with a hardwired controller, we found that the optimizations we made to MAGIC to make protocol processing efficient were crucial in keeping the performance loss small. These optimizations include separating the control and data transfer logic, providing support for speculative memory operations, extending the PP instruction set architecture and making it a dual-issue processor, and extensive pipelining between all units on the chip. We have shown that turning off these optimizations has a large adverse effect on overall performance.

Although the flexibility of FLASH results in some performance loss, it also provides a mechanism to detect and alleviate performance problems. By taking advantage of flexibility to optimize the protocol and directory structures, we believe FLASH can be competitive with any real hardwired design. In the future, we would like to perform a detailed comparison of FLASH with a real hardwired implementation. However, this requires either the availability of an actual hardwired machine or a very detailed paper design that accounts for deadlock issues and other practical concerns.

Acknowledgments

We would like to acknowledge the cooperation of Intel Corporation, Supercomputer Systems Division. This work was supported by ARPA contract N00039-91-C-0138. Mark Heinrich and Joel Baxter are supported by National Science Foundation Fellowships. John Heinlein is supported by an Air Force Laboratory Graduate Fellowship. Kourosh Gharachorloo is supported by Digital Equipment Corporation’s Western Research Laboratory. Mendel Rosenblum is supported by an National Science Foundation Young Investigator Award.

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.

[DEK+92] Todd A. Dutton et al. The Design of the DEC 3000 AXP Systems, Two High-performance Workstations. *Digital Technical Journal*, volume 4, number 4, pages 66-81. Digital Equipment Corporation, Maynard, MA, 1992.

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

[HGD+94] John Heinlein et al. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

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

[Intel94] Edmund A. Reese et al. A Phase-Tolerant 3.8GB/s Data-Communication Router for a Multiprocessor Supercomputer Backplane. In *Proceedings of the 1994 International Solid-State Circuits Conference*, pages 296-297, San Francisco, CA, February 1994.

[KOH+94] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, Chicago, IL, April 1994.

[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 156-167, 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.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325-336, Chicago, IL, April 1994.

[RSG93] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity for Large-Scale Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 14-25, San Diego, CA, May 1993.

[RV94] Mendel Rosenblum and Mani Varadarajan. SimOS: A Fast Operating System Simulation Environment. Technical Report CSL-TR-94-631, Stanford University, July 1994.

[Simoni92] Richard Simoni. Cache Coherence Directories for Scalable Multiprocessors. Ph.D. Thesis, Technical Report CSL-TR-93-556, Stanford University, November 1992.

[Smith92] Michael David Smith. Support for Speculative Execution in High-Performance Processors. Ph.D. Thesis, Technical Report CSL-TR-93-556, Stanford University, November 1992.

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

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5-44, March 1992.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.