Processor Mechanisms for Software Shared Memory

by

Nicholas Parks Carter

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January, 1999

[February 1999]

Author ....
Department of Electrical Engineering and Computer Science
January 25, 1999

Certified by ....
Professor William J. Dally
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Dr. Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Processor Mechanisms for Software Shared Memory

by

Nicholas Parks Carter

Submitted to the Department of Electrical Engineering and Computer Science on January 25, 1999, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

This thesis describes and evaluates the effectiveness of four hardware mechanisms for software shared memory: block status bits, a global translation lookaside buffer, a fast, non-blocking, event system, and dedicated thread slots for software handlers. These mechanisms have been integrated into the M-Machine's MAP processor, and accelerate tasks which are common to many shared-memory protocols, including detection of remote memory references, invocation of software handlers, and determination of the home node of an address.

The M-Machine's mechanisms for shared memory require little hardware to implement, including 3KB of RAM and the register files for the thread slots allocated to shared-memory handlers. Integrating these mechanisms into the processor instead of providing shared-memory support through an off-chip co-processor reduces the hardware cost of shared memory, eliminates inter-chip communication delays in interactions between the CPU and the shared-memory system, and improves resource utilization by allowing shared-memory handlers to use the same processor resources as user programs.

Hardware support for shared memory significantly improves the M-Machine's remote memory access time, allowing remote memory requests to be resolved in as little as 336 cycles, as compared to 1500+ cycles on an M-Machine without hardware support. In program-level experiments, the M-Machine's shared-memory system was shown to allow efficient exploitation of parallelism, achieving a speedup of greater than 4x on an 8-node FFT.

The MAP chip's non-blocking memory system was shown to be a significant contributor to the remote memory access time, as operations which reference remote data must be enqueued in a software data structure while they are being resolved. To improve performance, additional mechanisms, including transaction buffers, have been proposed and evaluated. In concert, the additional mechanisms proposed in this thesis reduce the remote memory access time to 229 cycles, improving program execution time by up to 16%.

Thesis Supervisor: William J. Dally
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This Ph.D. program has been a long and difficult process, and I owe a great deal of thanks to the many people who made it possible. First and foremost, I would like to thank the friends who kept me sane and reminded me that there's more to life than my job. To Anna, Jill, Tim, Mahk, Pete, Sara, Andy, the gang at Decmo, and the Twinkie Towers crowd, I'd like to say "Thanks. I couldn't have done it without you."

My family has also been a tremendous help throughout my life, and during my graduate school years in particular. Mom, Dad, I want to thank you for always being there when I needed you, and for accepting the times when what I needed was to be left alone to work it through by myself.

I've learned a great deal by watching the other members of the M-Machine project, including Whay Lee, Steve Keckler, and Andrew Chang. Professor Dally, my advisor, deserves credit for daring to start a project as ambitious as the M-Machine, and for his understanding when reality and our dreams proved incompatible.

A number of agencies provided the funding and equipment that made this work possible, including the Defense Advanced Research Projects Agency which funded the M-Machine project under contract F19628-92-C-0045, and my research in particular through an AASERT fellowship. I'd also like to thank Sun Microsystems for their generous donations of computing equipment to the CVA group, without which I'd probably still be waiting for my simulations to complete.

Finally, I'd like to thank whoever wrote the fortune cookie message which reminded me, at the height of the M-Machine implementation crunch, that "Every great accomplishment is at some point impossible."

Table of Contents

Chapter 1

Introduction

This thesis describes the M-Machine's shared memory system, which is designed to efficiently support a wide variety of shared-memory protocols through a combination of hardware and software. Four hardware mechanisms for shared memory have been integrated into the M-Machine's MAP processor: block status bits, a global translation lookaside buffer, a fast event system, and dedicated thread slots for shared-memory handlers. These mechanisms accelerate tasks which are common to many shared-memory protocols, such as detecting remote memory references, determining the home node of an address, and invoking software handlers, allowing the software handlers to focus on tasks which vary between protocols.

Integrating hardware support for shared memory into the processor provides significant improvements in remote memory access times at a low hardware cost. The shared-memory protocol implemented for this thesis is able to resolve a remote memory reference in as little as 336 cycles, more than an order of magnitude faster than software-only shared memory systems such as SHASTA [32], and less than 2.5x the remote access time of the SGI Origin 2000 [20], a contemporary system which implements its shared-memory protocol completely in hardware.

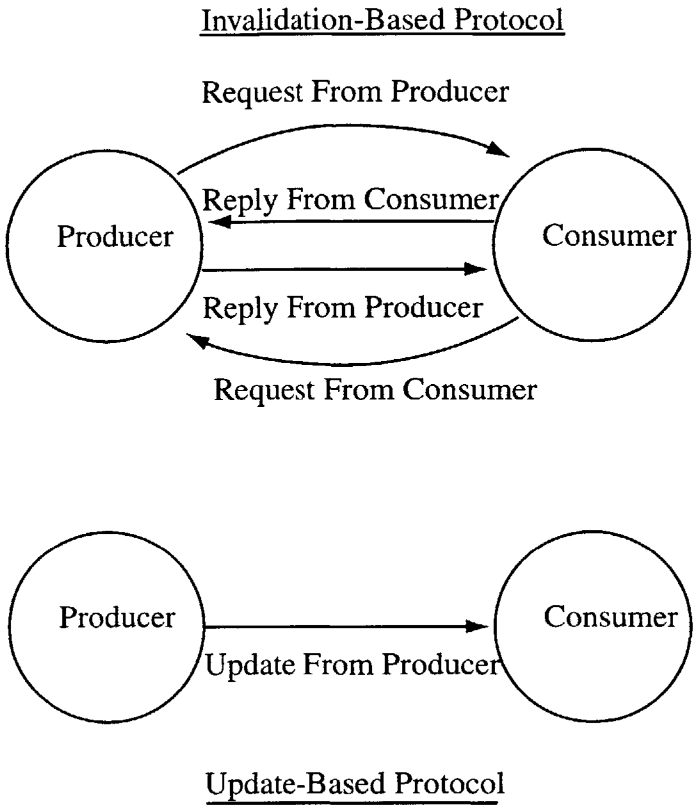
Implementing the M-Machine's mechanisms for shared memory requires less than 3 KB of RAM for the event system, global translation lookaside buffer, and block status bits, and 1.25 KB of register files for the dedicated thread slots used by the software handlers. In contrast, most other systems which provide hardware support for shared memory do so either through an off-chip memory management unit or a co-processor which executes shared memory protocols, both of which have significantly higher hardware costs than the

M-Machine's mechanisms, and also add inter-chip communication delays to the time required for any interaction between the CPU and the shared-memory hardware.

1.1 Why Flexible Shared Memory?

Shared-memory multicomputers have been successful in large part because of their relative ease of programming when compared with message-passing multicomputers, particularly for irregular problems. Because they allow inter-processor communication to be expressed in terms of references to shared data structures, many programmers find shared-memory systems easier to program than message-passing systems, which require that each inter-processor communication be explicitly specified by the programmer. However, this reduction in programming effort comes at the cost of limiting the programmer to the communication protocol implemented by the shared-memory system, while programmers of message-passing multicomputers can structure the communication patterns of their applications to suit the needs of their programs.

If the communication patterns of a program do not match the assumptions of the shared-memory protocol, limiting programmers to a single communication protocol can significantly reduce performance, because no single communication protocol handles all communication patterns well. An example of this can be seen in Figure 1.1 and Figure 1.2, which illustrate how update-based and invalidation-based shared memory protocols handle single-word and multi-word producer-consumer communication. In the single-word case, the update-based protocol is more efficient, because only one message per communication is required to transmit the producer's data to the consumer, while the invalidation-based protocol requires at least four messages: a request-reply pair for the producer to acquire the data to be written from the consumer, and a request-reply pair for the consumer to re-acquire the data in order to read the data that the producer has written.



Invalidation-Based Protocol

Update-Based Protocol

Figure 1.1: Single-Word Producer-Consumer Communication

In contrast, the invalidation-based protocol is more efficient if the producer generates multiple words of data to be read by the consumer, since it can take advantage of spatial locality by transferring a multi-word block of data between processors in response to each remote reference. Because of this, the invalidation-based protocol still requires four messages for each producer-consumer communication as long as the number of words written by the producer is less than the block size of the protocol, while the update-based protocol requires one message per word written by the producer, which becomes less efficient as the number of words increases.

Flexible shared-memory systems like the M-Machine close the efficiency gap between shared-memory and message-passing systems through architectures which support multi-

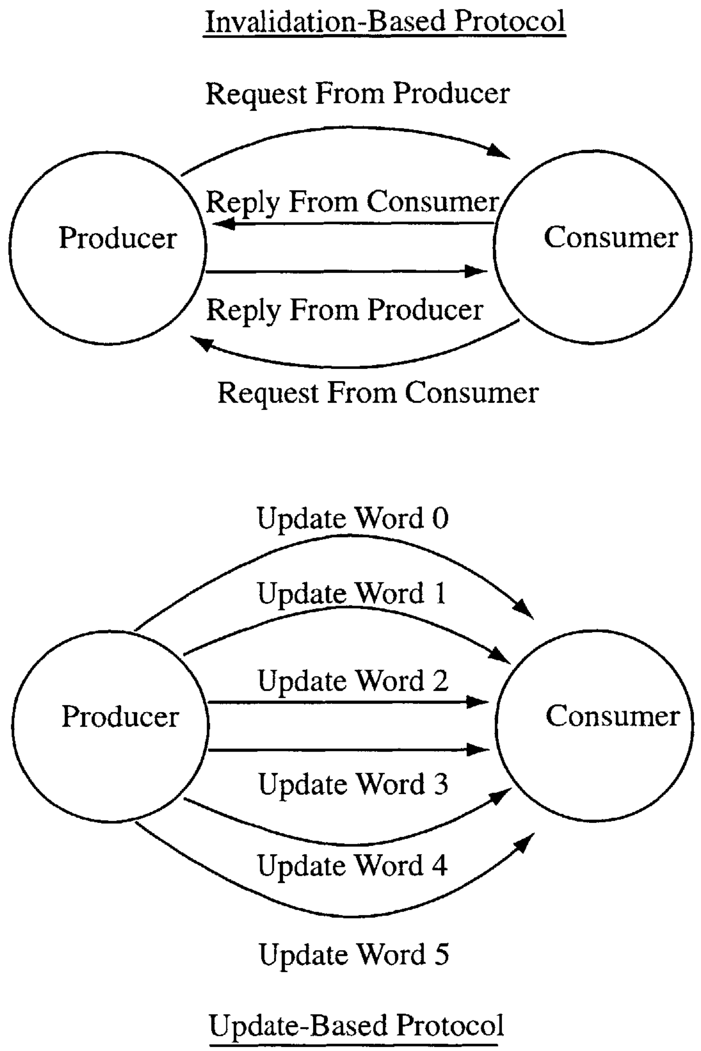


Figure 1.2: Multi-Word Producer-Consumer Communication

ple shared-memory protocols, allowing the programmer to select the best protocol for the application. However, flexible shared-memory systems generally have longer remote access times than single-protocol shared-memory systems because of their reliance on software as part of the shared-memory system in order to support multiple shared-memory protocols. Because of this, flexible shared-memory systems can achieve better performance than single-protocol systems on applications which are a poor match for the single-protocol system's shared-memory protocol, but suffer from their longer remote memory access times when running programs that are a good match for the single-protocol system's protocol. The M-Machine's hardware support for shared memory is intended to reduce this drawback by allowing tasks common to many shared-memory protocols to be performed at hardware speeds, reducing the amount of work that the shared-memory handlers have to do and thus the time required to execute the shared-memory handlers.

1.2 Contributions of This Thesis

The M-Machine's implementation of shared memory differs from previous approaches in that its hardware support for shared memory is integrated into the CPU of each processing node, taking advantage of the multithreaded architecture of the MAP chip to execute shared-memory handlers on the same processing resources that are used by user programs. This approach greatly reduces the amount of hardware required to support shared memory efficiently while still providing a great deal of flexibility. The M-Machine's hardware support for shared memory consists of four mechanisms which accelerate functions that are common to many shared-memory protocols. Two block status bits are associated with each 8-word block of data on a node, allowing individual blocks to be marked present or not present on the node, and thus allowing individual blocks to be transferred between nodes. Block status bits are maintained for each block on the node, not just the blocks in the cache, allowing remote data to be stored in the off-chip memory of the node to support "all-cache" protocols.

The Global Translation Lookaside Buffer caches translations between virtual addresses and node ID's, reducing the time required to determine the home node of an address. A novel encoding scheme is used to allow a single entry in the GTLB to map multiple pages of data across multiple nodes, greatly reducing the number of GTLB entries required. To detect remote accesses and invoke software handlers, the M-Machine uses an event system based around an *event queue* that allows a processing node to respond very quickly to operations that require the assistance of system software and allows event handlers to execute simultaneously with user threads. Finally, the M-Machine dedicates a number of thread slots on each node to running the system software used to implement shared memory. This eliminates the need to perform a context switch at the start of each handler, greatly reducing the remote memory access time.

To evaluate the effectiveness of these hardware mechanisms, a conventional cached shared-memory protocol has been implemented on the MSIM simulator for the M-Machine, and a number of microbenchmarks and application programs run in simulation. Variants of the shared-memory protocol have been written which disable individual mechanisms or combinations of mechanisms, allowing the effectiveness of each mechanism to be gauged.

Based on the evaluation of the M-Machine's mechanisms, three additional hardware mechanisms for shared memory have been designed and evaluated in simulation: hardware generation of configuration space addresses, instructions which ignore block status bits, and transaction buffers. In combination, these three mechanisms reduce the remote memory access time from 336 cycles to 229 cycles, an improvement of 32%. In keeping with the philosophy of the M-Machine's original mechanisms for shared memory, the hardware cost of these new mechanisms is small. Hardware generation of configuration space addresses and instructions which ignore block status bits require only a small amount of control logic to implement, while the transaction buffers require approximately 6KB of RAM for the 32-entry transaction buffer simulated for this thesis.

Even with all of the proposed new mechanisms for software shared memory, the M-Machine's remote memory access time is still more than 50% greater than that of contemporary shared-memory systems which implement a single shared-memory protocol completely in hardware. The future work section of this thesis outlines the design of a full-hardware flexible shared-memory system which makes use of a programmable state machine to implement multiple shared-memory protocols at hardware speeds and uses prediction to match the protocol to the needs of the application.

1.3 Thesis Roadmap

The remainder of this thesis begins with a discussion of previous work, showing the spectrum of approaches that have been used to implement shared memory in the past. After that, Chapter 3 describes the architecture of the M-Machine Multicomputer, including its mechanisms for on-chip and multi-node parallelism. Chapter 4 follows up this discussion with a detailed description of the M-Machine's mechanisms for shared memory and an illustration of how these mechanisms are used to implement various shared-memory protocols. Chapter 5 evaluates the M-Machine's mechanisms for shared memory, showing their impact on remote memory access times and program performance. Chapter 6 discusses the design of additional hardware mechanisms to improve shared-memory performance, while Chapter 7 provides a detailed comparison between the M-Machine and a number of other flexible shared-memory systems. Finally, Chapter 8 presents a high-level design for a full-hardware flexible shared-memory system which would provide the benefits of flexibility without the performance penalties incurred by using software to resolve remote memory requests. An appendix presents a detailed description of the shared-memory protocol implemented for this thesis.

Chapter 2

Previous Work

In order to provide a context for the discussion of the M-Machine's support for shared memory, this chapter briefly describes a number of previous shared-memory systems, showing examples of the spectrum of techniques which have been used to implement shared memory in the past. A more detailed comparison between the M-Machine and a subset of these shared-memory systems is presented in Chapter 7.

2.1 Centralized Shared-Memory Systems

Many of the early shared-memory computers, such as the IBM 370 model 168 and the CDC Cyber 170 [31], implemented shared memory by allowing multiple processors to access a centralized memory through a communication network, as shown in Figure 2.1. A variety of communication networks and memory system architectures were used by these machines, but their defining characteristic is that all memories are equally distant from all processors.

Bus-based shared-memory systems, such as the Berkeley SPUR machine [38], and the Firefly workstation [37], are an important subclass of centralized shared-memory systems, in which processors communicate with the centralized memory system over a shared memory bus. This architecture maintains cache coherence by having the memory controller for each processor examine each transaction on the memory bus. If the data requested is contained in the processor's cache, the processor takes over the memory bus to complete the transaction, ensuring that the requesting processor sees the most recent copy of the data.

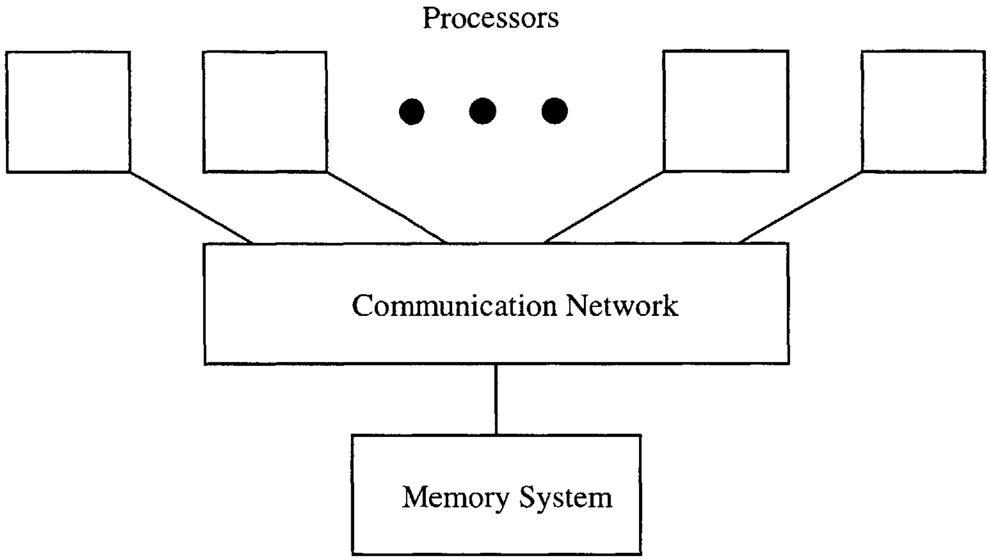


Figure 2.1: Centralized Shared-Memory Systems

2.2 Distributed Shared-Memory Systems

Centralized shared-memory systems are relatively simple to implement, and offer good performance for small numbers of processors. However, accessing the centralized memory becomes a performance bottleneck as the number of processors, and thus the delay across the communication network, increases. To combat this effect, designers have implemented *distributed* shared-memory (DSM) systems, in which each processor has a local memory, and communicates with the others via a communication network, as shown in Figure 2.2. In a distributed shared-memory system, regions of address space are assigned to the memory of each processor, which is then referred to as the *home node* of that address space. The home node is responsible for maintaining a consistent image of the memory it is responsible for, generally by using a *directory* to track which other nodes have copies of its data.

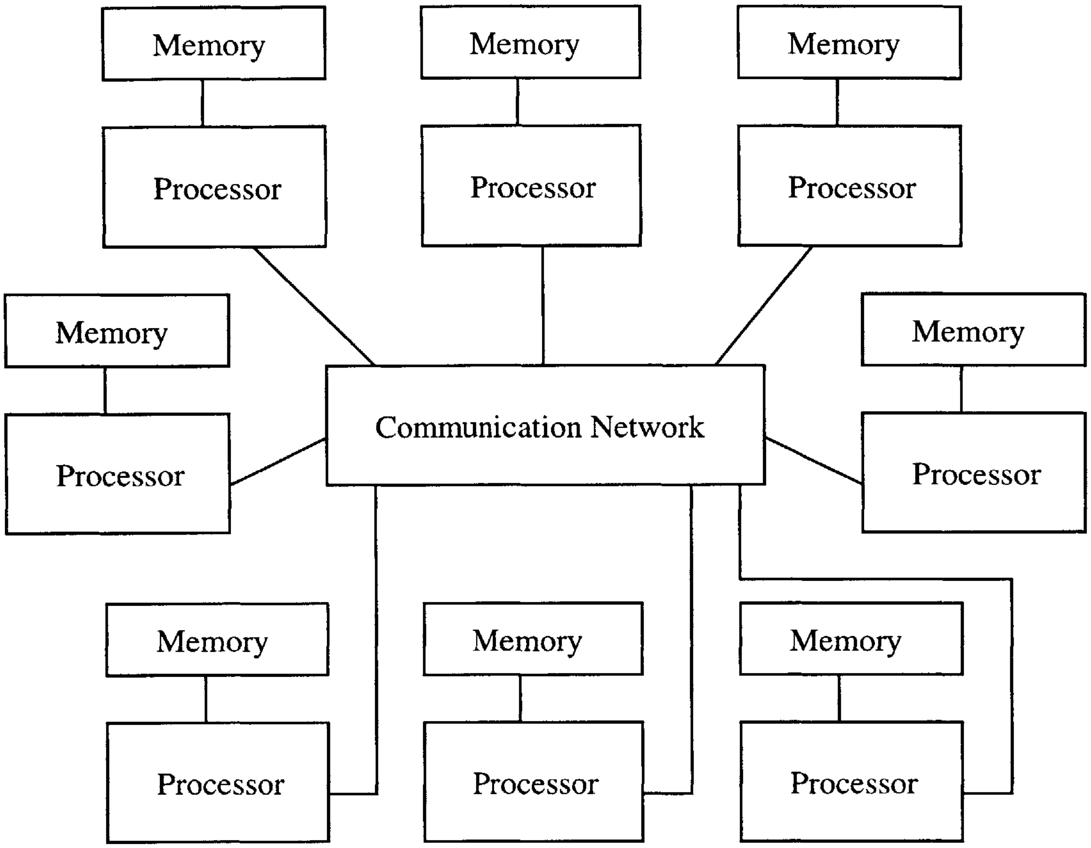


Figure 2.2: Distributed Shared-Memory System

Distributed shared-memory systems often have longer remote memory latencies than centralized shared-memory systems, due to the need to determine which node is the home node for a remote address, request the remote data, access the directory on the home node, and return the data to the requesting processor. However, they offer the ability to reduce memory latencies by assigning data to the local memory of the processor which references it most often, allowing that data to be accessed without traversing the communication network. In addition, distributed shared-memory systems can often sustain higher total remote memory bandwidths than centralized systems by allowing each home node to handle requests for its data in parallel and by replacing broadcasts of invalidation messages with point-to-point communications between the home node and the nodes with copies of the data. For these reasons, large distributed shared-memory systems tend to have better performance than equally-sized centralized systems, while the reverse is often true for systems with small numbers of processors.

2.2.1 Software-only Systems

Distributed shared-memory systems have used a wide variety of techniques to implement shared-memory, ranging from software-based approaches to machines which implement shared memory completely in hardware. For example, IVY [24], one of the first distributed shared-memory systems, implemented shared memory in software running on an Apollo ring network. User-level extensions to the operating system migrated pages of data between processors in response to memory requests, taking advantage of the hardware's virtual-memory system to force traps to software on references to invalid pages.

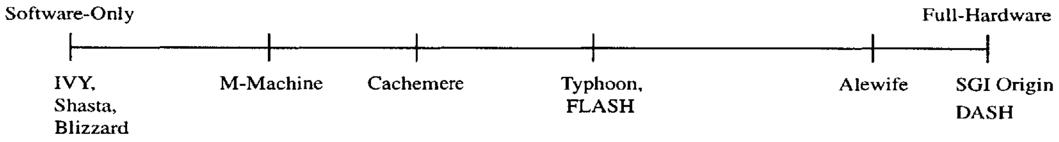


Figure 2.3: Spectrum of Shared-Memory Implementations

One major weakness of IVY was the false sharing which resulted from the large grain size of the shared-memory system, which could not move blocks of data smaller than the operating system's page size between processors. To address this limitation, Shasta [32] and Blizzard-S [33] modify program executables to add checking code before each remote memory reference which determines whether or not the referenced data is present on the node. This approach allows the shared-memory system to trade off block granularity against the amount of data required to record the state of the blocks resident on a node, at the cost of increased application run time due to the time required to perform the access checks.

Blizzard-E [33] takes another approach, making use of the CM-5's memory error correction features to mark fine-grained regions of data as present or not present on a node. This avoids the overhead of performing access checks in software on each memory reference, at the cost of requiring a privileged operation to modify the ECC bits during block installation and eviction. In addition, additional overhead is incurred when read-only and writable blocks are contained within the same page, as the system is unable to mark regions of data smaller than a page as read-only, causing spurious software traps when writing writable blocks within a page that contains read-only data.

2.2.2 Hardware-Only Systems

In contrast to these software-only approaches, DASH [22] [23] used a full-hardware scheme to maintain cache coherence. Each node contained a fully-mapped directory controller in addition to its processor and memory which tracked the data that the node was responsible for, using a bit vector to record which nodes had copies of each block of data. This simplified the design of the directory controller, but limited the maximum number of nodes that the system could support to the number of the bits in the bit vector.

To address this limitation, the SGI Origin [20] uses a multiple-granularity directory scheme which is implemented completely in hardware. On small systems, or if all of the nodes which share copies of a line reside within the same 64-node octant of a large machine, each bit in the directory bit vector corresponds to one node (2 processors). On larger systems, each bit in the vector corresponds to eight nodes (16 processors), allowing an Origin to support up to 512 nodes (1024 processors).

2.2.3 Hardware/Software Systems

The MIT Alewife Machine [1] uses a different approach to overcome the scalability limitations of fully-mapped directory scheme. The CMMU chip implemented for Alewife implements the LimitLESS [6] protocol, which allows up to five sharing processors for each block to be tracked in hardware, and traps to software to resolve requests to blocks which are shared by more than five processors. This allows the Alewife to support a large number of processors while still providing good performance, as most blocks are shared by fewer than five processors.

In the last several years, a number of systems have appeared which use a combination of hardware and software to implement shared memory in order to support multiple shared-memory protocols efficiently. The Tempest [29] [30] project at the University of Wisconsin-Madison showed that substantial performance improvements could be obtained on many programs by selecting a shared-memory protocol which matched the needs of the application, and developed the Teapot [7] language to simplify the process of implementing different shared-memory protocols. The Tempest effort also explored a number of methods of implementing flexible shared memory, including the Blizzard systems which were mentioned earlier and the Typhoon systems, which explored the performance tradeoffs involved in using different co-processor architectures to execute shared-memory protocols.

The Stanford FLASH [13] [14] machine also implements flexible shared memory through the use of a co-processor to execute shared-memory protocols. FLASH is based on the SGI Origin architecture, and replaces the cache-coherence controller with a custom protocol processor known as MAGIC. The MAGIC chip has been optimized for execution of shared-memory protocols, and is able to perform a simple remote read request in 111 100 MHz cycles, which compares well with the 140 196-MHz cycles required to complete a simple remote access on the SGI Origin.

2.3 Hybrid Shared-Memory Systems

In recent years, symmetric multiprocessing (SMP) systems, which typically use bus-based cache-coherence techniques to implement small-scale shared-memory systems out of commodity microprocessors, have become popular, and a number of projects have explored techniques for connecting multiple SMP processors together to build larger-scale multiprocessors. The MGS [39] effort, which arose out of the Alewife project, examined the performance tradeoffs involved in constructing a multicomputer where each node of the multicomputer contained multiple processors. The Cachemere [36] project constructed a multiprocessor out of SMP nodes using a network of multiprocessor workstations connected by a Memory Channel bus. Cachemere used a two-level cache-coherence policy, which used a page-based software scheme to maintain coherence between SMP nodes while relying on the SMP hardware to maintain coherence within each node. This two-level cache-coherence protocol showed significant performance improvements over using only the software coherence scheme for some applications, although others showed little or no improvement.

Chapter 3

The M-Machine Multicomputer

3.1 Motivation

We have designed and built the M-Machine to demonstrate an architecture which addresses two significant limitations of current computer architectures: their inability to exploit fine-grained parallelism, and their over-reliance on global communication. To address these limitations, the M-Machine uses a clustered processor architecture that reduces wire lengths, allows exploitation of instruction-level and fine-grained task-level parallelism within a single processor, and provides mechanisms for low-latency communication between processors. This combination of features allows the M-Machine to effectively exploit parallelism at all granularities, and should allow future processors based on the M-Machine's architecture to run at higher clock rates than conventionally-designed processors.

Inability to exploit fine-grained parallelism is a serious weakness of current computer systems. High inter-processor communication latencies limit parallel speedups, while instruction-level parallelism is limited by the number of independent instructions in sequential programs, leaving a *parallelism gap* between the two regimes in which today's systems effectively exploit parallelism. To obtain maximum performance on a wide spectrum of programs, future systems will need to close this parallelism gap by providing low-latency communication mechanisms which allow small tasks with large communication requirements to be parallelized effectively.

The M-Machine Multicomputer [11] addresses this parallelism gap through an architecture which effectively exploits parallelism at all granularities. An M-Machine consists of a two-dimensional mesh of processing nodes, each of which is made up of a custom

Multi-ALU Processor (MAP) chip and five synchronous DRAM (SDRAM) chips. To exploit instruction-level and thread-level parallelism, the MAP chip uses a clustered processor architecture which divides the on-chip functional units into three independent processor clusters. Processor clusters are programmed in a VLIW fashion to allow exploitation of instruction-level parallelism within a cluster, and communicate with each other over two on-chip switches and through the shared cache. These inter-cluster communication mechanisms allow the programmer to treat the three clusters as independent processors to exploit thread-level parallelism, or as a single VLIW, to exploit instruction-level parallelism. To provide latency tolerance and fast invocation of message and event handlers, each processor cluster is multithreaded, with two user and three system thread contexts per cluster.

An on-chip network unit on the MAP provides low-latency user-level messaging between processing nodes, allowing coarser-grained parallelism to be exploited across multiple nodes of an M-Machine. Programmers have the option of writing message-passing programs which explicitly specify inter-node communication, or of writing shared-memory programs which rely on system software to orchestrate data movement between nodes.

In addition to providing mechanisms for exploitation of parallelism at multiple granularities, the MAP chip's processor architecture also reduces the amount of communication required in a clock cycle when compared with more conventional architectures. Superscalar and VLIW processors rely on centralized register files and instruction issue logic to exploit instruction-level parallelism, which requires that each arithmetic unit communicate with the centralized resources on each clock cycle. In contrast, the MAP chip's clustered processor architecture allows each cluster to operate independently, reducing wire lengths and signal propagation delays. Since wire delay is expected to become the dominant contributor to processor cycle times within a few process generations [34], this reduction in global communication should allow future processors based on the MAP to achieve higher clock rates than those based on superscalar or VLIW techniques.

3.2 MAP Chip Architecture

Figure 3.1 shows a block diagram of a MAP chip. The MAP contains three processor clusters, two cache banks, an external memory interface, and a network subsystem. Communication between the various subsections of the MAP chip occurs over three switches: the memory switch (M-Switch), the cluster switch (C-Switch), and the external memory bus (X-Bus). The M-Switch carries memory requests from the clusters to the cache banks, while the C-Switch carries replies back to the clusters from the memory system, and allows direct communication between threads running on different clusters. The X-Bus handles the communication between the cache banks and the external memory interface.

The MAP chip has been implemented in a 0.5 micron (0.7 micron drawn) process with five layers of metal. Chip tape-out occurred during the summer of 1998, and the first MAP chips were available in October of 1998. Work on the system board is ongoing, and assembly of the prototype M-Machine is expected early in 1999.

3.3 Cluster Architecture

Each of the MAP chip's processor clusters acts as an independent, multithreaded, processor, which allows large numbers of functional units to be integrated onto a single microchip without requiring global communication between the functional units on each cycle. Figure 3.2 shows a block diagram of cluster 0, which contains three functional units: one integer, one memory, and one floating-point[1]. Each cluster also contains an instruction cache and connections to the C- and M-Switches to allow communication with the other

[1] Clusters 1 and 2 contain only integer and memory units. The floating-point units were deleted from these clusters during implementation of the MAP chip due to area constraints.

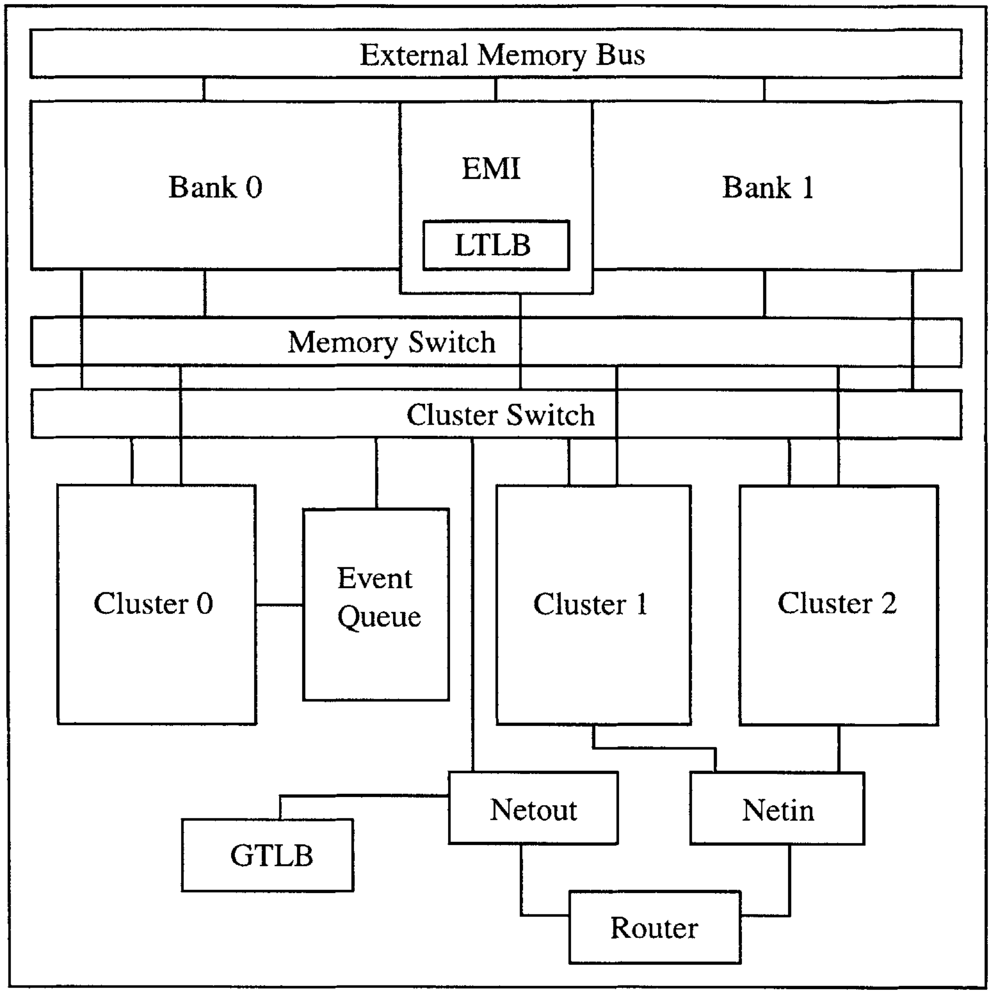


Figure 3.1: MAP Chip Block Diagram

modules on the MAP chip.

Clusters are programmed using a VLIW instruction format, in which each instruction specifies an operation to execute on each of the functional units in the cluster. This allows exploitation of instruction-level parallelism within a cluster without the complex control logic of a superscalar design. A variable-length instruction encoding is used which encodes NOP operations as single bits to reduce the amount of instruction bandwidth consumed by unfilled operation slots.

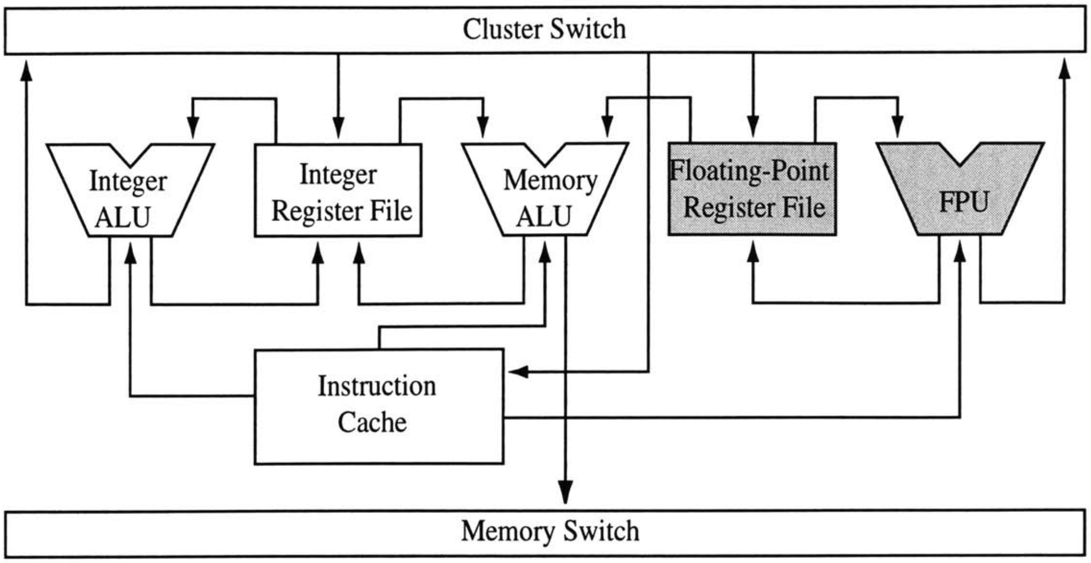


Figure 3.2: Cluster 0 Block Diagram. Shaded components are unique to cluster 0, while unshaded components are found in all clusters.

3.3.1 Multithreading for Latency Tolerance and Parallelism

The MAP chip's multithreading scheme is based on the Processor Coupling model [17] which was developed during the early stages of the M-Machine project, with a number of modifications which further decouple the clusters to reduce global communication. On each cycle, each of the clusters independently examines the programs running in its five thread slots, known as H-Threads, and selects an instruction to issue based on operand and instruction availability. This allows zero-cycle multithreading between the threads running on each cluster, eliminating the context switch overhead found on many earlier multithreaded processors, such as the Alewife SPARCLE [2].

To implement zero-cycle multithreading, the MAP chip replicates the instruction decode and register read stages of the pipeline for each thread slot and adds an additional pipeline stage, known as the synchronization stage, to the pipeline, as shown in Figure 3.3. Instructions from each of the thread slots flow through the pipeline in parallel until they reach the synchronization stage, which makes the decision about which instruction to issue on each cycle. The implementation of the MAP chip's multithreading scheme is described in more detail in [8] .

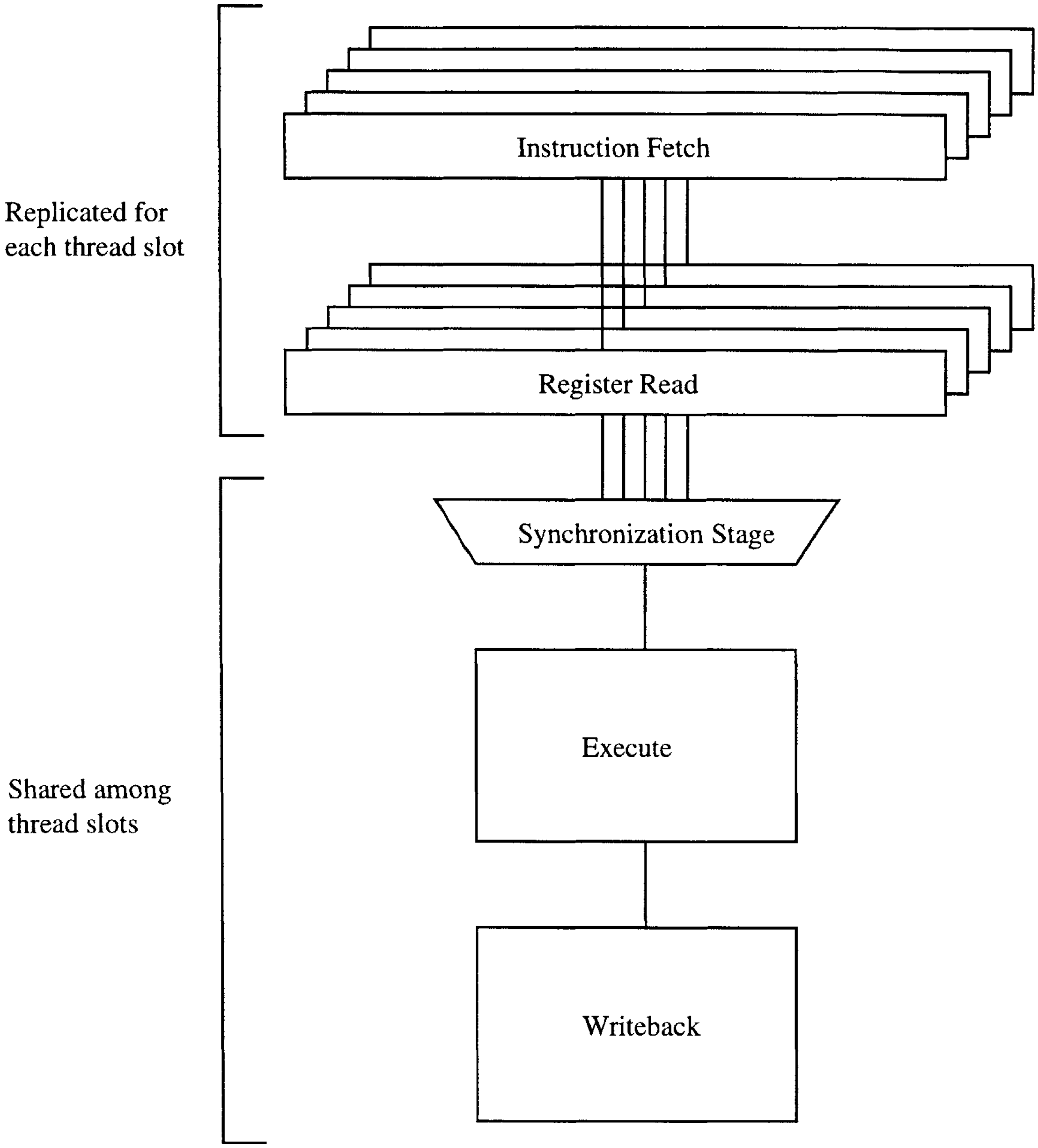


Figure 3.3: MAP Chip Pipeline

If no additional inter-thread communication mechanisms existed on the MAP, it would still be an effective single-chip multiprocessor, using the shared cache for inter-thread communication. To provide faster communication between threads, the MAP groups the threads running in the same thread slot on each of the clusters into a thread group, known as a V-Thread, as illustrated in Figure 3.4. The H-Threads that make up a V-Thread are allocated a thread slot on each cluster as a gang, and are allowed to write directly into each other's register files by specifying a remote register file as the destination of any operation, incurring a one-cycle latency penalty to do so. While the H-Threads that make up a V-Thread must reside in the same protection domain, since they can modify each other's register state, V-Threads from multiple protection domains may be active on a MAP simultaneously. The MAP's Guarded Pointer protection scheme, which will be discussed later in this chapter, provides memory protection which is independent of address translation, allowing the MAP to issue instructions from V-Threads which reside in different protection domains simultaneously.

To reduce communication between clusters, the MAP chip does not automatically empty the destination register of an operation which target's another cluster's register file when the operation issues, as it does with operations which target the local register file. Instead, the **EMPTY** instruction is used to invalidate registers for inter-cluster communication. The **EMPTY** instruction takes one argument, a bit vector of registers to be emptied, and marks all of the registers specified in the vector empty when it issues.

Figure 3.5 shows an example of inter-cluster communication on the MAP. First, both of the clusters execute **EMPTY** operations, marking the destination registers for the inter-cluster communication invalid. Then, each cluster executes a **CBAR** (Cluster BARrier) operation, synchronizing the two threads, and guaranteeing that both clusters have executed their **EMPTY** operations. Once this has been done, the clusters can then communi-

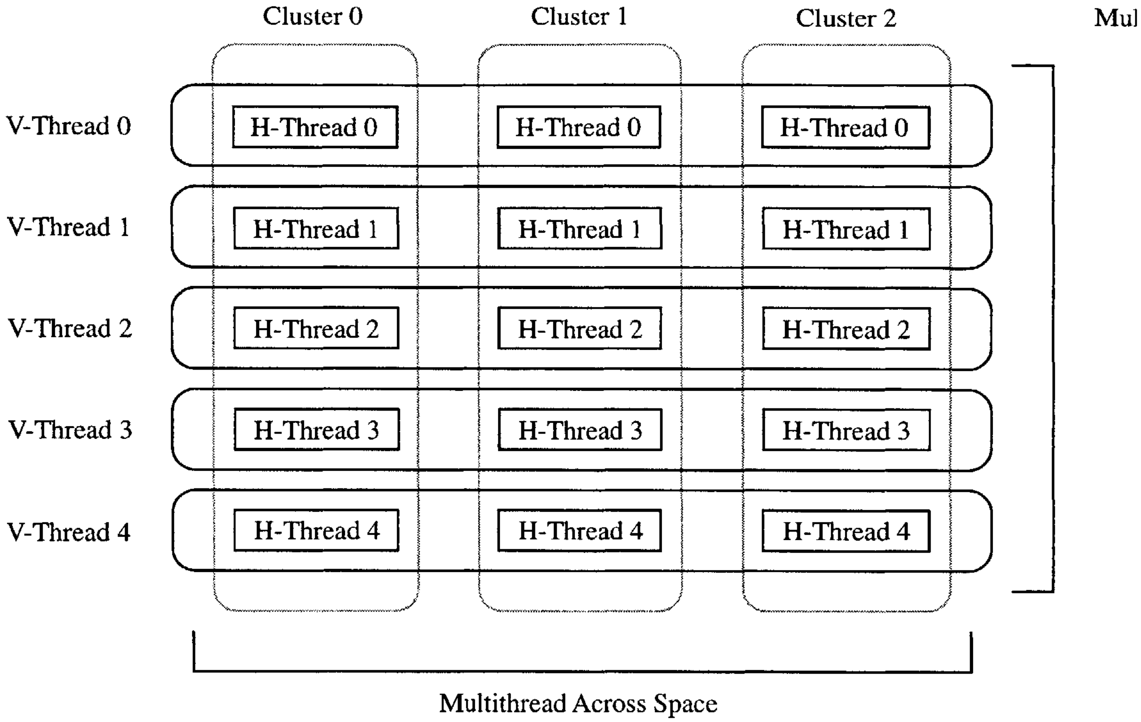


Figure 3.4: Multithreading Scheme

cate by first emptying the target register in their register file, and then writing data into the register file of the other cluster. As long as each cluster references its target register before emptying it, this protocol keeps the clusters synchronized without extraneous communication.

This support for inter-cluster communication allows the MAP to be programmed in two distinct manners. The first approach is to treat each of the clusters as an independent processor, using the inter-cluster communication mechanisms to reduce the communication and synchronization overhead. This approach has been shown [18] [16] to be very effective, achieving noticably higher speedups then memory-based communication on a number of fine-grained applications.

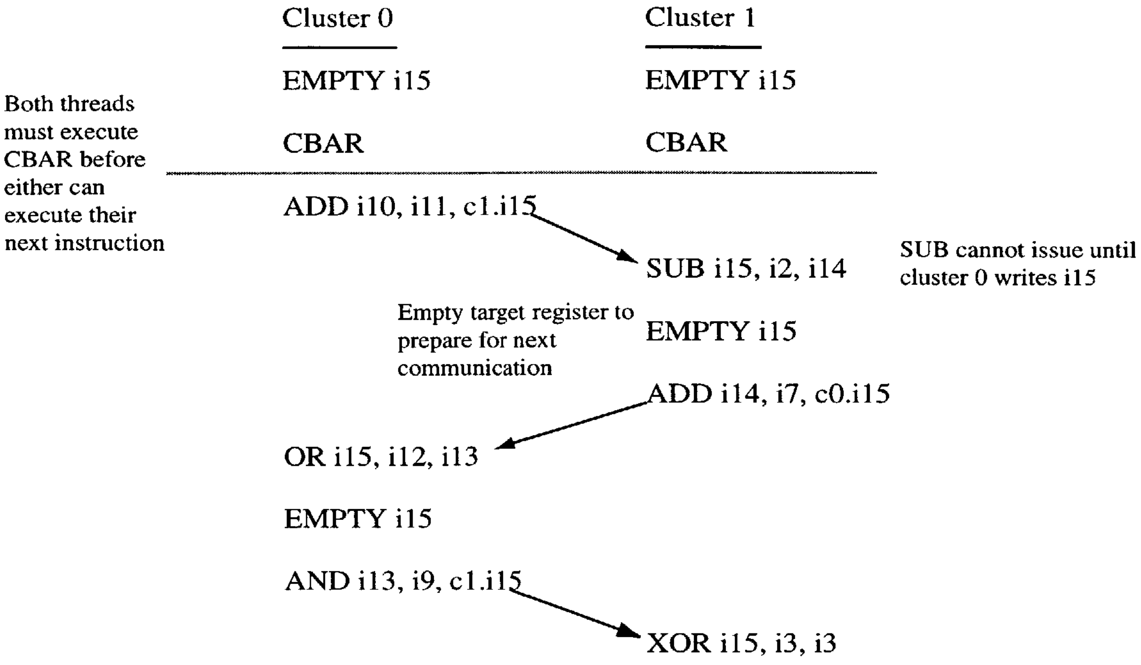

Cluster 0
Cluster 1
EMPTY i15
EMPTY i15
Both threads must execute CBAR before either can execute their next instruction
CBAR
CBAR
ADD i10, i11, c1.i15
SUB i15, i2, i14
SUB cannot issue until cluster 0 writes i15
Empty target register to prepare for next communication
EMPTY i15
ADD i14, i7, c0.i15
OR i15, i12, i13
EMPTY i15
AND i13, i9, c1.i15
XOR i15, i3, i3

Figure 3.5: Inter-Cluster Communication

The second programming methodology for the MAP treats the three clusters as a large VLIW, and relies on the compiler to schedule communication between the clusters. This approach has the advantage that it requires little effort on the part of the programmer to parallelize programs, but the effectiveness of the MAP when programmed in this fashion has not been conclusively demonstrated. Research [26] has shown that the overhead involved in managing data transfers between clusters is small, and work is ongoing to develop compiler technology to program the MAP in this fashion.

Another way that the MAP's multithreading scheme contributes to its performance is the reduction in event and message handling time which results from dedicating thread slots to specific handlers. Figure 3.6 shows how the thread slots on the MAP are allocated. Threads 0 and 1 on each cluster are allocated to user threads, while threads 2-4 execute system threads. Thread 2 contains the exception handlers, thread 3 contains the event handler and two reserved system thread slots, while thread 4 contains the TLB miss handler and the message handlers. As shown in Chapter 5, providing dedicated threads for the event and message handlers reduces the remote memory latency on the M-Machine by about 90 cycles by eliminating the need to perform a context switch every time a handler is invoked.

Thread	Cluster 0	Cluster 1	Cluster 2
0	User 0	User 0	User 0
1	User 1	User 1	User 1
2	Exception	Exception	Exception
3	Event Handler	System	System
4	TLB Miss Handler	Priority 0 Message Handler	Priority 1 Message Handler

Figure 3.6: Thread Allocation on the MAP

3.4 Switch Design

The C-Switch and M-Switch allow communication among the processor clusters and between the processor clusters and the cache banks. The C-Switch is a 7-input, 3-output crossbar, implemented as three data busses (one for each cluster), with arbitration between the 7 input sources (three clusters, two memory banks, the external memory interface, and the configuration space controller). The M-Switch uses a similar implementation, using two busses, one for each cache bank, and arbitrating among the requests from each of the three clusters.

Both the M-Switch and the C-Switch allocate 1/2 cycle for flight time along their data busses. The C-Switch requires that drivers arbitrate for the switch on the cycle before they wish to transfer data so that the data arrives in time to allow the synchronization stage in the receiving cluster to issue an instruction which depends on the data in the next cycle.

The M-Switch arbitrates during the first half-cycle and transfers data during the second half-cycle, as the cache banks do not need their input data until the end of the cycle.

Because of the long wires (18+ mm) contained in the switches, careful circuit design was required to enable high-speed operation without violating the current limits imposed by the process. The switch drivers use a "break before make" technique to reduce the switching current by guaranteeing that there is never a path between VDD and ground through the switches. Future processors based on the MAP will need to use additional circuit techniques, such as low-voltage signalling, to avoid violating process current limits, and may have to increase the delay through the switches as fabrication processes change.

3.5 Memory System

The MAP chip's processor architecture requires that the memory system be able to execute multiple memory operations per cycle, while providing memory protection in spite of the fact that the memory operations executed in a given cycle may come from multiple threads executing in different protection domains. To meet these requirements, the MAP chip incorporates a banked cache subsystem and a protection scheme based on Guarded Pointers [5], a novel implementation of capabilities that overcomes many of the problems with previous capability-based systems. Guarded Pointers separate protection from translation, allowing independent threads to safely share a single address space.

3.5.1 Cache Architecture

To resolve multiple memory accesses per cycle, the MAP chip includes a dual-banked main cache, which acts as an L1 cache for data and an L2 cache for instructions, backing the instruction caches in the clusters. Each cache bank contains its own tag and data arrays, as shown in Figure 3.7, and addresses are interleaved between the banks on a word-by-word basis. Memory requests are sent to the cache over the M-Switch, and replies are sent back to the clusters over the C-Switch when a cache hit occurs. If a cache miss occurs, the request is forwarded to the EMI over the X-Bus, which resolves the request and sends the reply back to the requesting cluster over the C-Switch. The X-Bus is also used to transfer data between the cache banks and the EMI as part of the process of reading and writing lines from the off-chip memory.

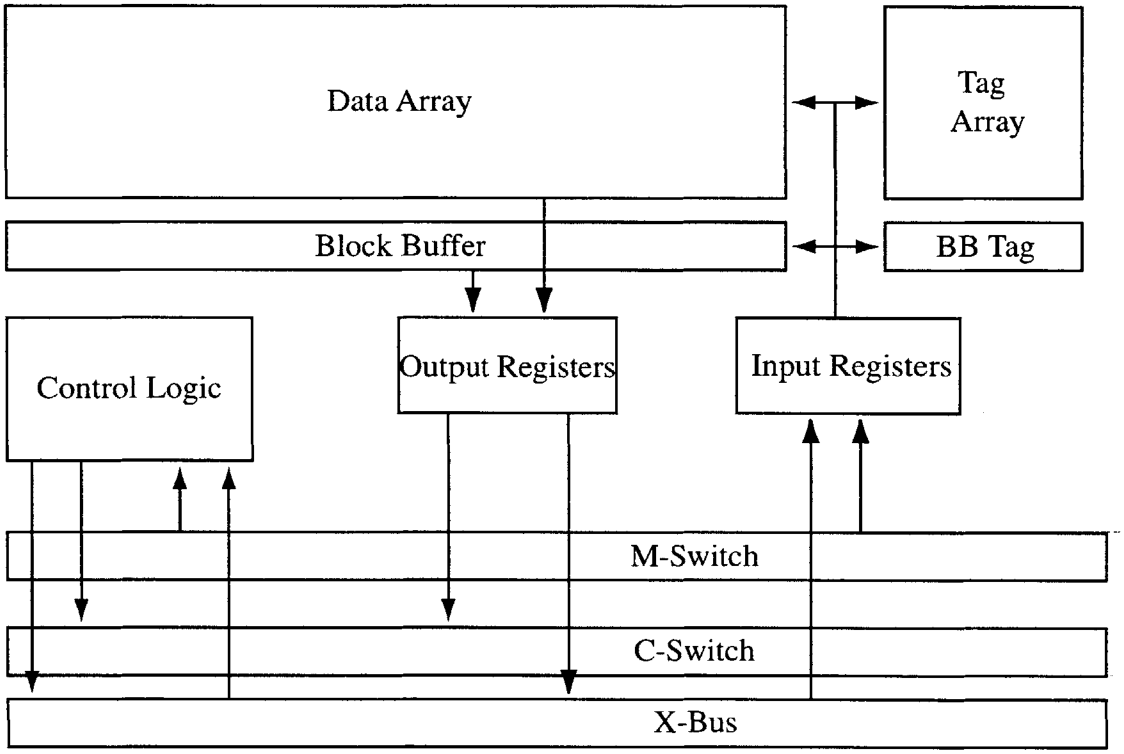


Figure 3.7: Cache Bank Block Diagram

One unusual aspect of the cache banks is the presence of the block buffer, a two-line buffer which acts as a victim cache [15], although our implementation differs from Jouppi's original design in that lines in the block buffer are not returned to the main cache if a hit to the block buffer occurs. When lines are evicted from the main cache, they are copied into the block buffer in a single cycle, reducing the cache miss latency. Cache lines remain accessible while they are in the block buffers, reducing the impact of conflict misses in the two-way set-associative main cache. Additionally, the load uncached (LUC) instruction allows a cache line to be fetched directly into the block buffers from the main memory to avoid conflicts with data already in the cache. Another thing to note about the cache bank architecture is that there is no connection to the TLB, as the Guarded Pointers protection scheme does not require that virtual addresses be translated to provide protection. This allows the MAP to use a fully virtually-addressed cache and only perform address translation on cache misses, reducing the complexity of the TLB and facilitating sharing of data between threads.

There were a number of interesting trade-offs made during the design of the cache banks. One of these was the decision to implement instruction caches in each of the processor clusters, but not data caches. Implementing L1 instruction caches in each cluster is necessary in order to achieve good performance, as the peak instruction bandwidth required by each cluster is 1.5 words per cycle, leading to a peak bandwidth requirement of 4.5 words/cycle in order to allow the three clusters to issue at their maximum rate. Since the peak bandwidth available from the memory system is 2 words/cycle, local instruction caches are necessary to prevent instruction bandwidth from being the limiting factor on performance.

Implementing L1 data caches in each cluster would have improved performance for two reasons. First, it would have reduced the cache hit latency, by eliminating the round trip flight time over the switches, and second, it would have increased the data bandwidth available to the clusters. However, implementing local data caches in each cluster would have required some mechanism for keeping the multiple caches coherent, which would have increased the complexity of the memory system design, a problem which does not occur with the read-only local instruction caches. Implementing local instruction caches but a shared data cache provides sufficient instruction bandwidth for the clusters, and guarantees that all of the clusters see the same memory image, at a cost of adding one cycle to the cache-hit latency. Future processors based on the MAP architecture will probably need to implement local data caches either in each cluster or shared by a subset of the clusters, as increasing wire delay will make it impractical to access a single shared cache for each data reference.

3.5.2 External Memory Interface

The external memory interface (EMI) is responsible for controlling the off-chip memory and orchestrating the transfer of data between the cache banks and the SDRAMs. As shown in Figure 3.8, the EMI contains the local translation lookaside buffer (LTLB), as well as a set of input and output registers.

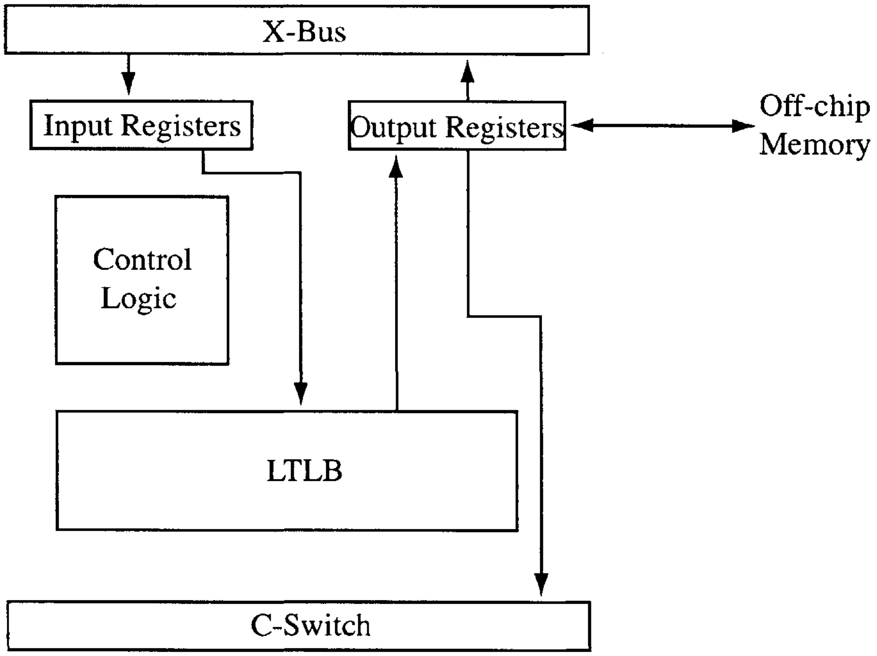


Figure 3.8: EMI Block Diagram

The LTLB is a relatively conventional 64-entry, direct-mapped TLB, and is responsible for performing virtual-physical address translation whenever a cache miss occurs. In addition, the LTLB contains the block status bits for each block within the pages it maps, and forwards these bits to the cache banks when a line is brought into the cache. Chapter 4 describes the block status bits in more detail.

When a cache miss occurs, the cache bank that sustained the miss sends the request which caused the miss to the EMI over the X-Bus. The EMI then performs address translation on the address, and checks the block status bits of the line being referenced to ensure that the reference is allowed. If the address translation or block status check fails, the event system is invoked to start a software handler to resolve the problem. Otherwise, the EMI sends the proper control signals to the off-chip memory to fetch the missing line, and sends the line to the cache banks over the X-Bus. The operation which caused the cache miss is then resolved by the EMI, and the result sent back to the requesting cluster over the C-Switch.

3.5.3 Guarded Pointers

To provide memory protection in a multithreaded environment, the M-Machine implements a capability-based [10] protection system known as Guarded Pointers. Each word in the Machine's memory is tagged with an unforgeable tag bit, known as the pointer bit. If the tag bit is not set, the contents of the word are interpreted as an integer or floating-point value by the hardware. If the tag bit is set, the word is interpreted as a pointer, and special meaning is assigned to different sub-fields within the word, as shown in Figure 3.9. Four bits of the pointer indicate its type, which tells the hardware which operations may be performed using the pointer. Six bits identify the length of the segment containing the pointer, which is used by the hardware to divide the 54-bit address field into a segment field, which user operations may not change, and an offset field, which may be altered by user operations. This format allows a pointer to define an address, the region of data containing the address, and the set of operations that may be performed on the address, without requiring any software table look-ups to do so.

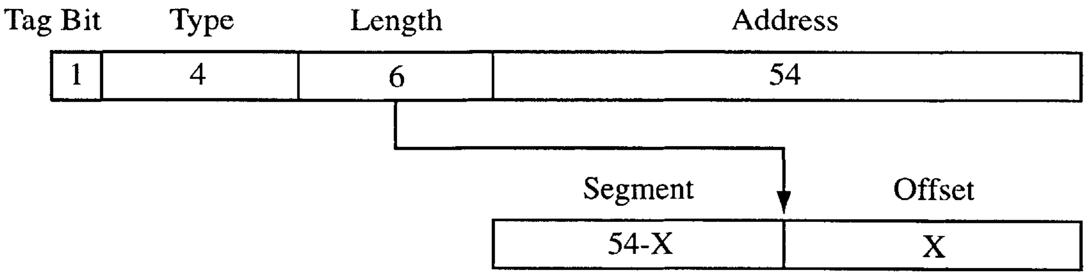


Figure 3.9: Guarded Pointer Format

Guarded pointers have a number of advantages over traditional translation-based protection for systems like the M-Machine. First, they allow multiple threads to share a single virtual address space in a protected fashion, as the set of data accessible to a thread is defined by the pointers contained in its register file and the pointers reachable through those pointers. Second, they facilitate controlled sharing of data between threads, as any threads with pointers to a region of data may access that data. Third, they simplify the memory system hardware by reducing the number of address translations required in each cycle.

One interesting use of Guarded Pointers on the M-Machine is the *Configuration Space*. Pointers with a particular value in their type field identify locations within the configuration space, which maps all of the internal registers on the MAP into an address space which is orthogonal to the main memory address space. The configuration space is also used for a number of other functions, such as controlling the MAP chip's I/O controller, and defining arbitrary transactions on the C-Switch. This last feature is used by the shared-memory software to complete remote memory accesses by simulating the C-Switch transaction which would have been generated had the operation completed without software intervention.

3.6 Network Subsystem

The M-Machine's network subsystem is designed to provide protected, low-latency, user-level messaging between nodes in an M-Machine, and consists of the network input and output queues, the GTLB, and the router. M-Machine nodes are connected in a two-dimensional mesh using dimension-order routing. Two message priorities (P0 and P1) are supported, to allow deadlock-free request-reply messaging.

Messages are composed in the register files of a thread, and then sent with an atomic **SEND** instruction. Upon arrival at the destination node, the router places the message in the appropriate network input queue for its priority. The head of each message queue is mapped into the register file of a dedicated thread slot, which allows the message handler thread to respond very quickly to incoming messages.

As discussed in [21], this network interface allows lower-latency messaging than architectures which map network control registers into their memory address space or which require explicit copying of messages into message buffers. It would have been possible to reduce message latency by using a streaming message output protocol, as was used in the J-Machine [28], but this has the disadvantage of allowing one thread to gain access to the network output unit for arbitrary periods of time, denying other threads use of this resource. By requiring that messages be composed in a thread's register file before being sent, the M-Machine limits the length of messages and thus the amount of time that any one thread can control access to the network output queue.

3.7 Events and Exceptions

The M-Machine divides exceptional conditions into two categories: *events* and *exceptions*. Events are conditions, such as TLB misses, which require software intervention to resolve but are part of normal program execution, while exceptions represent error conditions which generally require the termination of the program which caused the exception.

Events are handled by the event system, which is described in detail in Chapter 4.

To enable speculative execution of operations, the MAP implements a deferred exception mechanism, somewhat similar to the one used in Intel's EPIC architecture [3]. When most exceptions occur, the hardware generates a type of Guarded Pointer called an *ERRVAL*, which contains the address of the operation which caused the exception. ERRVALs may propagate through the program, as most instructions output a copy of any ERRVAL passed to them as an input, choosing one if more than one of their inputs are ERRVALs. Propagating ERRVALs in this way allows the programmer to determine where the first exceptional condition occurred, facilitating debugging.

When the MAP chip detects a condition which it cannot handle by generating an ERRVAL, such as an attempt to use an ERRVAL as the address of a store instruction, or an attempt to perform a comparison on an ERRVAL[1], an *exception* occurs. When an exception occurs, the hardware halts all user threads on the cluster on which the exception occurred, and invokes the exception handler thread on that cluster by writing into the thread's register file. The exception handler thread then resolves the operation, possibly by correcting the condition which caused the exception, but more often by displaying an error message and halting the thread.

Deferring exceptions by generating ERRVALs allows programs to schedule operations speculatively to improve performance. For example, a division operation can be scheduled in parallel with the check to see if the divisor is zero, since dividing by zero will create an ERRVAL rather than causing an exception. However, deferring exceptions can make it more difficult to recover from them, as significant time can elapse between the operation which caused a deferred exception and the detection of the operation.

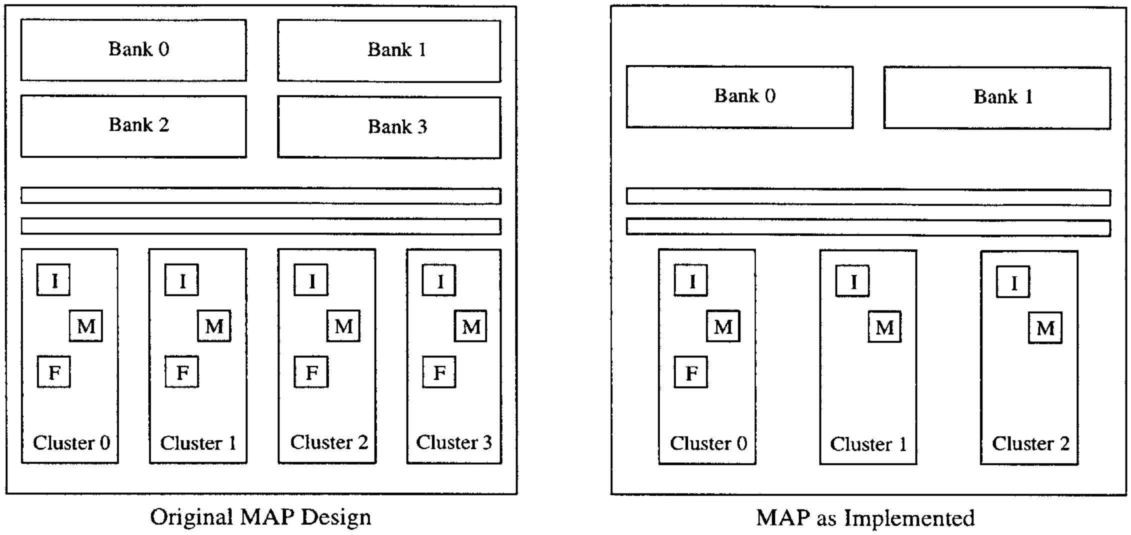
[1] Comparison operations output a one-bit condition code, which is too small to hold an ERRVAL.

The goal of the M-Machine's exception system was to allow the maximum possible speculation, while still providing enough information to allow the programmer to determine where the first exceptional condition occurred for debugging purposes, with the assumption that most exceptions would result in the termination of the program which caused them. To increase the number of exceptions which can be resolved without terminating the program which caused them, the M-Machine could use a software convention similar to the one used by EPIC, which requires that the results of all operations be tested to see if they contain exception values before any of their source operands are overwritten. This would guarantee that the exception handler had access to the inputs to any operation which caused an exception, but would reduce performance somewhat given the MAP chip's small (16-register) register files.

3.8 Evolution of the MAP Architecture

The MAP chip has undergone a number of changes since its architecture was first proposed in 1992. The original specification called for four processor clusters, each of which contained integer, memory, and floating-point execution units, four cache banks containing a total of 128KB of memory, six thread slots per cluster, and much larger TLBs than were implemented. During the implementation of the chip, area constraints forced the elimination of one of the clusters, two of the remaining floating-point units, and 75% of the cache memory, leaving two cache banks and three clusters, only one of which has a floating-point unit. Figure 3.10 shows block diagrams of the original MAP design and the MAP as implemented.

Two factors were responsible for the need to reduce the amount of memory and number of arithmetic units on the MAP chip: growth of control logic, and use of a cell-based implementation methodology for much of the cluster datapath. The control logic in the clusters required many more gates than had been anticipated during initial floorplanning,



Original MAP Design

MAP as Implemented

Figure 3.10: Evolution of the MAP Chip

partially due to the decision to move some functionality out of the datapaths and into the control logic to simplify the design of the datapaths. In addition, the CAD tools were unable to obtain the placement density that had been expected, partially due to the need to route several large data busses over the control region, reducing the number of available wiring tracks. As a result, the amount of the chip allocated to control logic had to be increased significantly over the original floorplan, reducing the area available for arithmetic units.

The cluster datapaths turned out to be significantly larger than had been anticipated, due to the decision to use a cell-based datapath design style instead of full-custom circuits for many of the datapath modules. Using the cell-based design methodology significantly reduced the effort required to implement the datapaths by allowing re-use of low level datapath cells, but increased the area required for the datapath modules implemented in this fashion by approximately 40%. The design effort, area, and performance impacts of the different design styles used on the MAP are discussed in more detail in [9].

The MAP chip's multithreading scheme also evolved significantly over the course of the project. Processor Coupling, as proposed in [17], treated the clusters within the MAP as a modified VLIW with 12 functional units. Individual operations within a 12-wide instruction could issue independently, but all of the operations in an instruction were required to issue before any operation from the next instruction could issue. Each cluster contained its own register files, requiring explicit communication operations to move data between clusters, but inter-cluster synchronization overhead was completely eliminated.

During implementation, the multithreading scheme was changed to the one described earlier, in which each cluster acts as an independent, multithreaded, processor, and threads running on different clusters have to execute explicit synchronization operations to maintain ordering. The **CBAR** operation and a tightly-coupled execution mode, which mimics the original Processor Coupling scheme by requiring that each of the H-Threads in a V-Thread execute an operation before any of them may execute a second operation, were also implemented to allow comparison of different methods of synchronization in single-chip multiprocessors. These changes simplified the implementation of the MAP, as each cluster now acts as a 3-wide VLIW, issuing all three operations within an instruction in the same cycle instead of allowing individual operations to issue in different cycles.

Steve Keckler's thesis [16] compared the three methods of inter-cluster synchronization available on the MAP: synchronization through memory, synchronization using the **CBAR** instruction, and implicit synchronization using the tightly-coupled execution mode. As expected, synchronization using the **CBAR** instruction was more efficient than synchronization through memory. More interestingly, synchronization using **CBAR** produced better results than using the tightly-coupled mode in some cases, as using **CBAR** operations allows greater latency tolerance by preventing long-latency operations on one cluster from delaying operations on other clusters unless there is a synchronization or data dependence between the operations. Thus, the relative performance of synchronization using **CBAR** and synchronization using tightly-coupled execution can vary depending on whether the increase in instructions due to the addition of **CBAR** operations or the increase in latency tolerance is more significant for the program in question.

3.9 MAP Chip Implementation

The MAP chip has been implemented in a 0.5-micron (0.7 micron drawn) process, with five layers of metal. It measures 18 mm on a side, and contains approximately 5 million transistors. A preliminary plot of the MAP chip is shown in Figure 3.11. Tape-out occurred in June of 1998, and first silicon was completed in October of 1998. When the system boards become available in the spring of 1999, a 16-node prototype M-Machine will be assembled to test the implementation and to run larger programs than has been possible in simulation.

3.10 Software Environment

The compiler and assembler for the M-Machine are based on the Multiflow compiler [25], which has been ported to the M-Machine. Currently, the compiler reliably generates single-cluster code which takes advantage of the multiple functional units on each cluster to improve performance. A version of the compiler exists which treats a V-Thread running on all three of the clusters in a MAP as a single VLIW by scheduling operations across all of the clusters and adding inter-cluster communication instructions as necessary. The multi-cluster compiler is not as stable as the single-cluster compiler, and therefore was not used for this thesis.

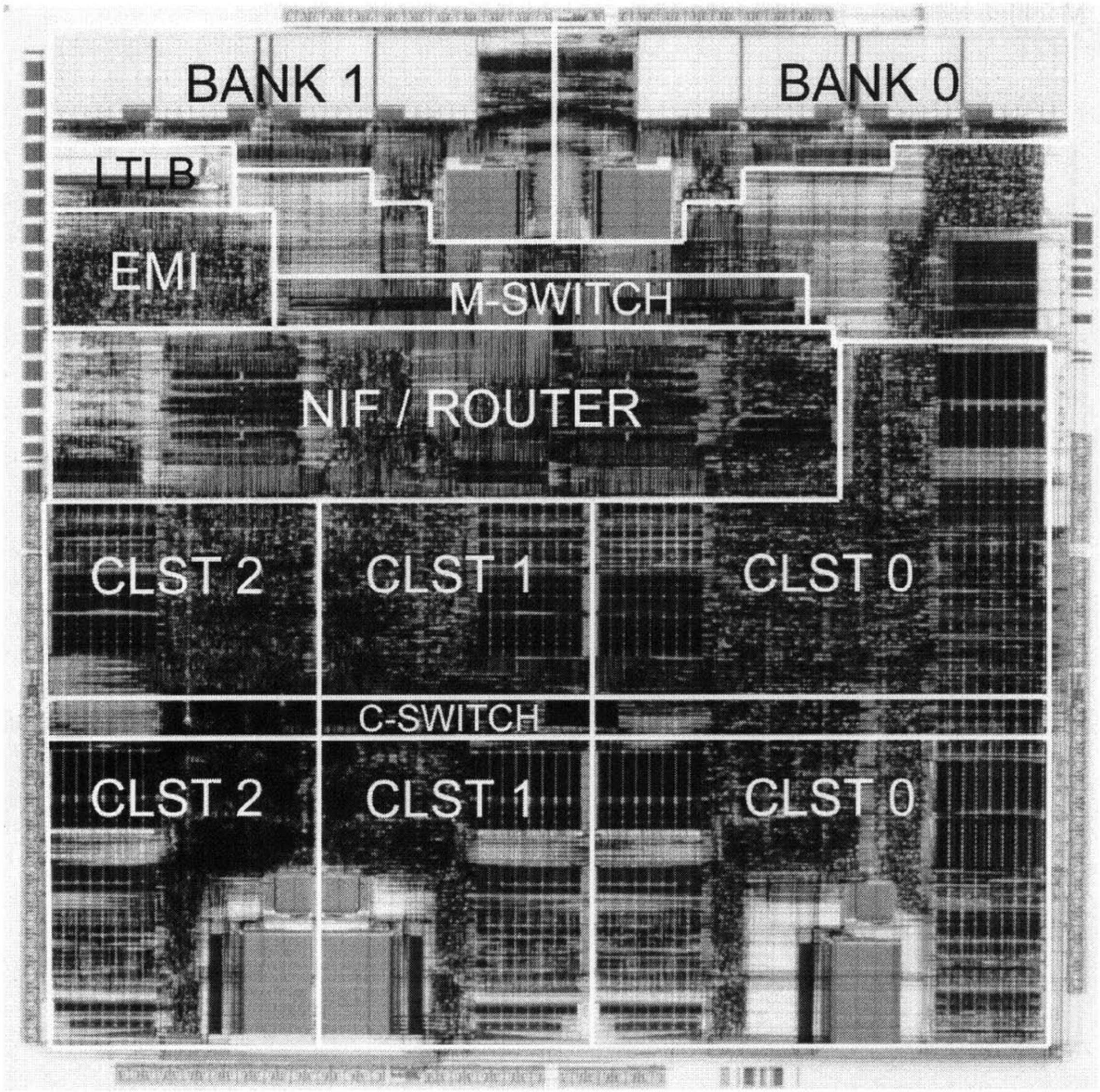


Figure 3.11: MAP Chip Plot

MARS, the runtime system for the M-Machine [12] [35], handles program loading and virtual memory management. In addition, MARS supports a number of UNIX operating system calls through a combination of program code running on the simulated M-Machine and "magic" C code in the MSIM simulator. This allows programs running on the simulator to print to the screen, read keyboard input from the user, and access file systems on the computer running the simulator, greatly simplifying the task of implementing programs on

MSIM, although these system calls will obviously have to be re-written once the M-Machine hardware becomes available.

Chapter 4

Mechanisms for Shared Memory

4.1 Philosophy/Goals

The M-Machine provides a number of communication mechanisms which allow message-passing programs to efficiently exploit inter-node parallelism, including user-level messaging and thread slots dedicated to processing incoming messages. However, the amount of effort required from the programmer to implement message-passing parallel programs is substantial, usually much greater than the effort required to implement a shared-memory program with the same functionality. To provide an intermediate point on the effort/program speed curve, it was decided that the M-Machine would support shared-memory programming models through a combination of hardware and software, allowing greater parallel speedups than would be possible on a single MAP chip, but requiring less effort from the programmer than implementing message-passing programs.

Using a combination of hardware and software to implement shared memory has a number of advantages for the M-Machine. First, it provides flexibility, allowing programmers to implement multiple shared-memory protocols and select a protocol which matches the needs of their application to improve performance. Second, it reduces the complexity of the MAP chip by eliminating the control logic that would be required for a full-hardware shared-memory system and also the need to verify the correctness of the cache-coherence protocol implemented by the hardware. Third, providing some hardware support for shared memory greatly reduces the remote memory latency at a low cost in both chip area and complexity. Finally, our approach to shared memory makes the M-Machine an attractive platform for research into cache-coherence protocols, as new protocols can be tested at near-hardware speeds without the effort involved in implementing the protocol in hardware.

To enable the M-Machine to support a wide variety of shared-memory protocols efficiently, we take advantage of the MAP chip's multithreading and protection mechanisms by dedicating a number of thread slots to the software handlers required by the shared-memory protocol. In addition, we provide hardware mechanisms to accelerate four subtasks which are part of most shared-memory protocols: detecting remote accesses, invoking software handlers to resolve remote accesses, determining the home node of an address, and transferring small blocks of data between nodes. These mechanisms substantially improve remote memory access times as compared to the M-Machine without hardware support for shared memory at a low hardware cost: approximately 3KB of memory and a small amount of control logic. Implementing these mechanisms also did not increase the amount of effort involved in verifying the MAP chip nearly as much as building a full-hardware cache-coherence system would have, because each of the mechanisms is relatively simple and can be tested independently of the others.

4.2 Hardware Mechanisms

In addition to the dedicated thread slots for shared-memory handlers, the MAP chip incorporates three mechanisms to improve the performance of shared memory: block status bits, which allow small blocks of data to be moved from node to node; the event system, which detects accesses to remote data and invokes handlers to resolve those accesses; and the Global Translation Lookaside Buffer, which performs translations between virtual addresses and the node ID's of the home nodes of those addresses.

4.2.1 Block Status Bits

One of the weaknesses of software-only shared-memory systems is that conventional virtual memory systems record the presence or absence of data on a node on a page-by-page basis, forcing the shared-memory protocol to either transfer an entire page of data to resolve each remote memory request, which results in substantial false sharing, or to add software checks to each memory reference to determine whether the address is local or remote, increasing the latency of both local and remote memory references. To avoid these difficulties, each 8-word block of data on an M-Machine node is annotated with two *block status bits*, which encode the states invalid, read-only, read-write, and dirty. The block status bits for each page are stored in the page table entry for that page, taking up 128 bits per entry (2 bits times the 64 8-word blocks in a 512-word page). When the translation for a page is installed in the LTLB, the block status bits for the page are copied into the LTLB. Similarly, the block status bits for each block are copied into the cache banks when the block is referenced.

Whenever a block of data is referenced, the block status bits are checked by the hardware, as shown in Figure 4.1 and Figure 4.2, to determine whether the status of the line allows the reference to complete. If the block status bits allow the reference, the hardware completes the operation and returns the result to the thread which made the reference. If not, the event system, which is described in the next section, is notified that the request requires software intervention to complete. The hardware automatically changes the status of a block from writable to dirty when the block is modified, and can be configured (via a configuration space operation) to demote blocks from readable to invalid if a program attempts to write the block. This last feature is intended to assist programmers in implementing cache-coherence protocols which provide strict memory ordering by preventing read operations which follow an attempted write from completing and seeing the old value of the data, but is optional to provide greater latency tolerance for protocols which do not require strict memory ordering.

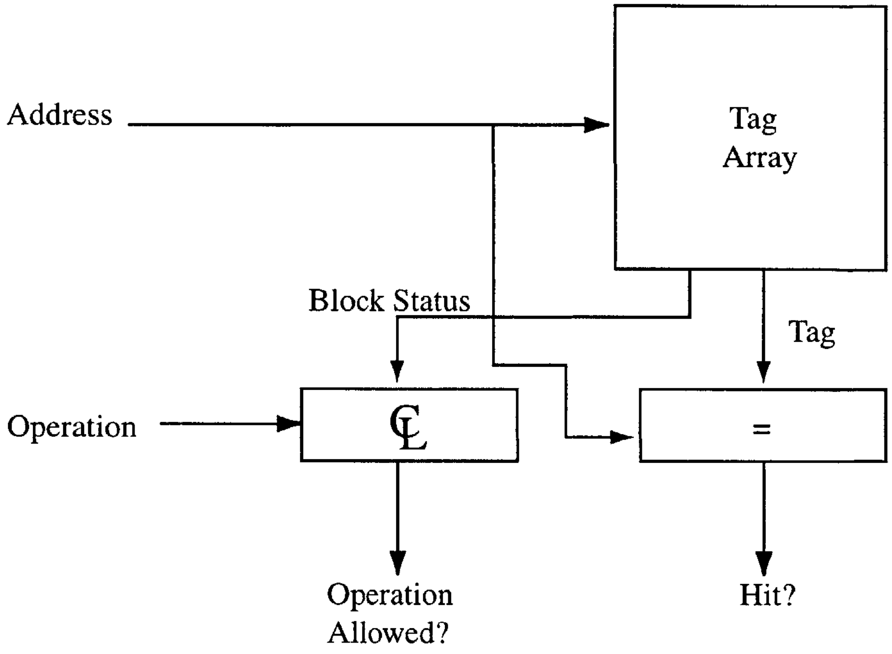


Figure 4.1: Block status bit check in the cache

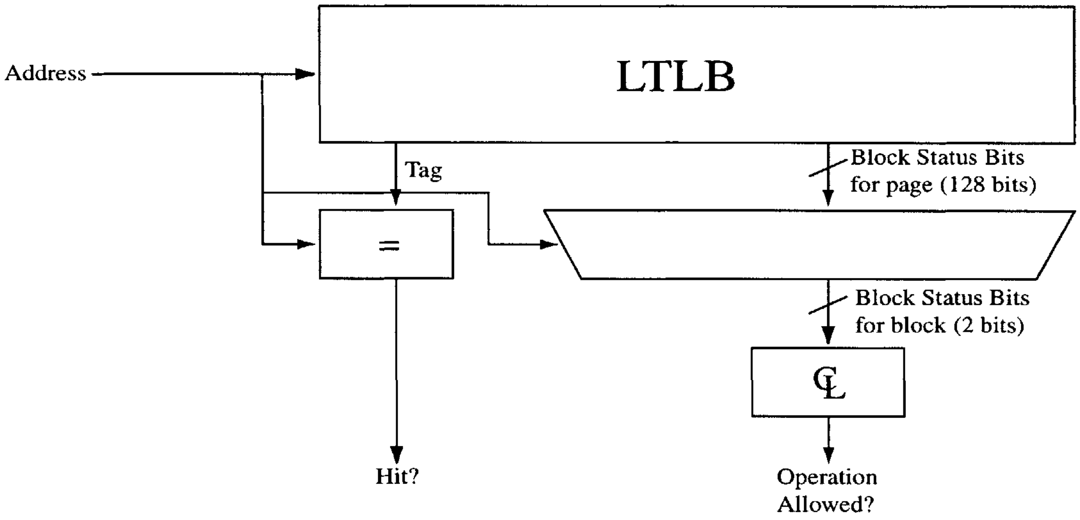


Figure 4.2: Block status bit check in the EMI

Implementing block status bits makes it very easy for the operating system to transfer individual blocks between nodes, at a low hardware cost. As shown in Figure 4.3, the first time a remote page (one whose home node is not the node performing the reference) is referenced, an LTLB miss occurs. The LTLB miss handler then creates a page table entry for the page with all of the block status bits set to invalid, which is installed into the LTLB. The hardware then retries the reference, and determines that a translation exists but that the data is not resident on the node, so the software shared-memory handlers are invoked. When the handlers have brought the block containing the referenced address onto the node, they set the block status of the block to either read-only or writable with a special **PUTCSTAT** instruction and complete the original request. Future references to the block can then be completed by the hardware, but references to any other block within the page will be detected as remote references because the block status bits of the referenced block are set to the invalid state.

Adding block status bits to the MAP chip required 1KB of SRAM to hold the 128 block status bits for each entry in the 64-entry LTLB, and 256 bytes of SRAM in the cache arrays (2 bits for each of the 512 blocks in the cache, replicated in both banks). Two 64:1 muxes are required in the LTLB to select the correct set of block status bits out of the bits for the entire page, and a small amount of control logic is required to perform the comparison between the operation being attempted and the block status of the referenced block.

4.2.2 The Event System

On the MAP chip, the event system is responsible for invoking software handlers in response to operations that require software intervention to complete, but that do not indicate an error condition or otherwise require the termination of the program which caused the event. References which are not permitted by the block status bits are resolved by the event system, as are GTLB misses and a number of other conditions. As shown in Figure

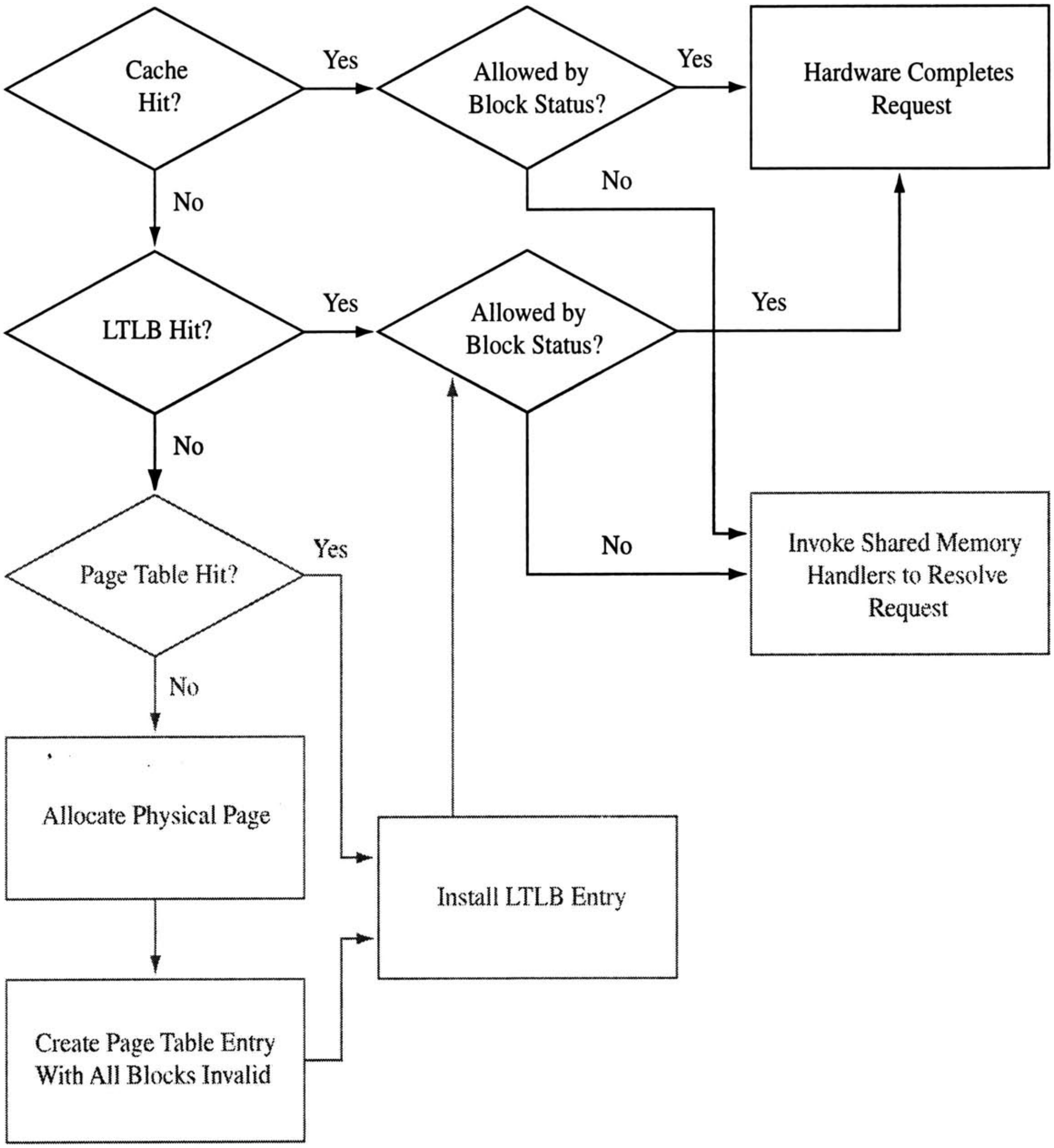


Figure 4.3: Memory Operation Flowchart

4.4, the event system consists of a 128-word[1] *event queue*, some control logic in the external memory interface and network unit, and a dedicated thread slot (thread 3 on cluster 0), where event handlers are executed.

When the control logic in the memory system detects an event, it creates a multi-word *event queue entry* which describes the event and sends the event queue entry to the event

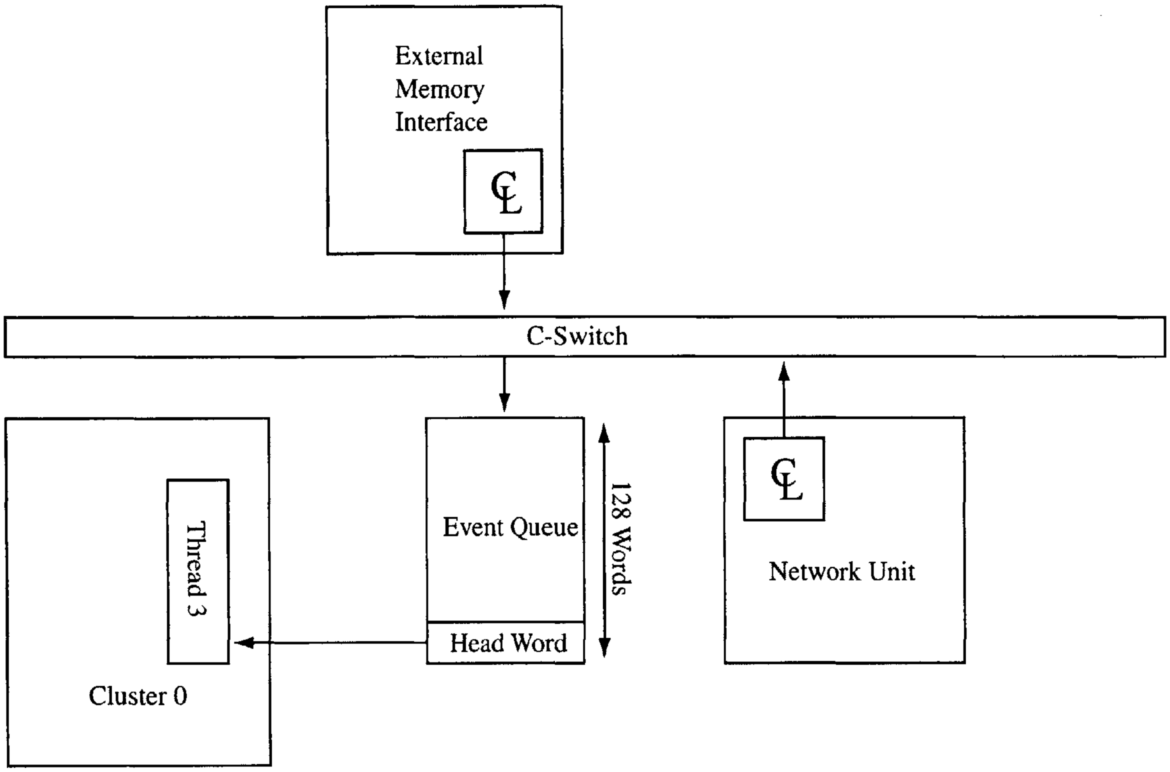


Figure 4.4: Event System Block Diagram

queue over the C-Switch. The event queue entry contains enough information to identify the event and, in the case of events generated by the memory system, enough information to re-generate the operation that caused the event. Once the event queue entry has been generated, the memory system discards the operation that caused the event, and is free to execute other operations from the same or different threads. Operations which cause events in other modules are handled similarly.

The head word of the event queue is mapped onto register 15 of the integer register file of the event thread slot, which is marked empty if there are no words in the event queue, and full otherwise. This provides for very fast invocation of event handlers without con-

[1] The size of the event queue was selected to be somewhat larger than the number of words required if every in-flight operation on the MAP were to take an event. To prevent the event queue from overflowing, the event queue disables execution of user threads whenever the free space in the queue drops below the amount required if all in-flight operations generate events.

suming execution resources in polling. When the event handler completes handling an event, it tries to read this register, to determine if there are any events left in the queue. If the queue is empty, the synchronization stage will detect the attempt to read an empty register and prevent the operation from issuing. When an event is placed in the queue, register 15 is marked full, and the synchronization stage adds the event handler thread to the set of threads that can issue an operation.

The event system allows software handlers to begin execution within 10 cycles of the execution of an operation which causes an event if the event handler is idle, since the event handler thread can begin execution as soon as the head word of the event queue is filled. Since the memory system discards the operation which caused the event as soon as the event queue entry has been enqueued, the thread which caused the event may continue execution while the event is being resolved until it attempts to use the result of the operation which caused the event. Other threads running on the MAP are not affected at all except for the few cycles during which the memory system hardware is busy generating the event queue entry.

The event system does not require a great deal of hardware. Some control logic is required in the memory system and the network unit, and several registers are required to compose the event queue entry. The event queue itself requires 1KB of SRAM[1], and some control logic. There is also an opportunity cost involved in dedicating a thread slot to resolving events that is hard to quantify.

4.2.3 The Global Translation Lookaside Buffer

In order to resolve a remote memory reference, it is necessary to determine the home node of the referenced address, so that a request for the address may be sent. To accelerate

[1] The event queue on the MAP was implemented using register cells for ease of implementation, but a commercial implementation of the MAP would probably use SRAM to save area.

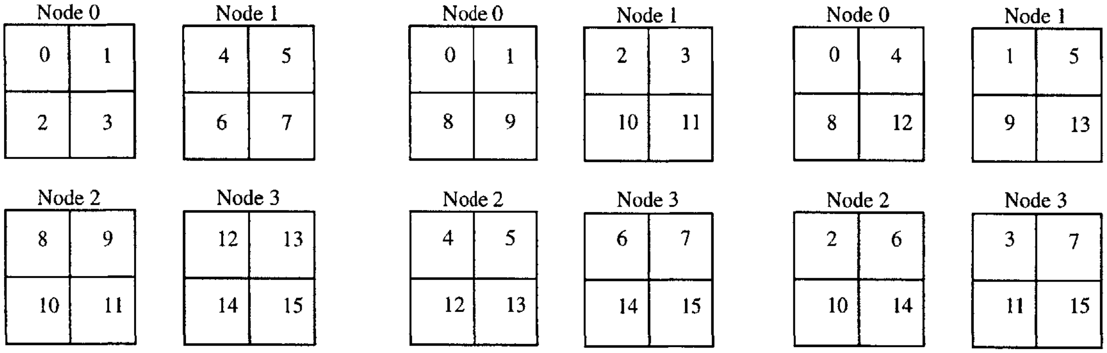
this process, we have implemented the Global Translation Lookaside Buffer (GTLB). The GTLB acts as a cache for translations between virtual addresses and node ID's, in a manner similar to the way a normal TLB caches translations between virtual addresses and physical addresses. Programs may access the GTLB in two ways. The first is by using a virtual address to specify the destination of a message, in which case the GTLB is implicitly accessed by the network system to determine the destination node of the message. The second is to use a **GPRB** instruction, which takes an address as its input and returns a node ID when it completes. This method is most useful for system programs, as user programs are prohibited from using node ID's to specify the destination of a message.

The GTLB differs from a conventional TLB in that it uses a novel mapping scheme which allows one GTLB entry to define how a variable-sized block of memory is mapped across a rectangular set of nodes within an M-Machine. Figure 4.5 shows the format of a GTLB entry. Each entry specifies the base address of the block of memory it maps, the size (in pages) of the block, the start node of the set of nodes that the memory is mapped across, the X- and Y-extents of the set of nodes that the memory is mapped across, and the number of consecutive pages to be mapped on each node. All of the fields except the base address and start node fields actually specify the base-two logarithm of the quantity in question, which reduces the size of a GTLB entry substantially, although it does limit the GTLB to only mapping blocks of memory which are a power of two pages in length across sets of nodes which are a power of two nodes on a side.

Base Address	Size	Start Node	X Extent	Y Extent	Pages Per Node

Figure 4.5: GTLB Entry Format

The pages per node field allows the memory mapped by a GTLB entry to be interleaved across the specified set of nodes at varying granularities, as shown in Figure 4.6, which illustrates the three ways in which sixteen pages may be mapped across a 2x2 set of nodes. If the pages per node field is set to the size of the region divided by the number of nodes, pages are mapped such that each node contains a contiguous set of pages. If the pages per node field is set to one, pages are mapped in a round-robin fashion, with each successive page mapping onto the next node, wrapping around to the start when necessary. Intermediate values of the pages per node field cause blocks of varying size to be interleaved across the nodes.



Pages per node = 4 Pages per node = 2 Pages per node = 1

Figure 4.6: Multiple Page Mappings

The GTLB's mapping scheme allows the MAP chip's four-entry GTLB to map all of the memory required by most programs. However, the flexibility of the GTLB increases the complexity of the logic required to perform an address translation. Because each entry in the GTLB maps a variable-sized block of data, the GTLB must be fully associative, as it is not possible to use a subset of the address bits to identify a set of entries which might contain the translation for the address. In addition, a masked comparator must be used to determine which bits of the address must be compared to the tag to determine if a hit has occurred. Figure 4.7 shows how the masked comparator works. The size field of the GTLB entry is fed into a mask generator, whose output vector has all zeroes in bit positions less than the value of the size field, and ones in the higher bit positions. Each bit of this vector is ANDed with the result of XORing the corresponding bits of the input address and the tag, which has the effect of zeroing any bit not required for the comparison. The resulting output bits are fed into a wide NOR which generates the hit/miss signal.,

Figure 4.8 shows the full process of performing a translation in the GTLB. Each entry is compared in parallel to the address being translated. If a hit occurs, the X- and Y-Extent fields of the entry which hit are used to select the bit fields of the address which contain the X- and Y-offsets from the starting node. The values of these bit fields are then added to the appropriate sub-fields of the base node ID to determine the home node of the address.

On the MAP chip, the GTLB is implemented using hand-placed standard-cell logic for simplicity of design and easy integration of the required storage and logic functions, and therefore takes up somewhat more chip area than would be required in a full-custom implementation. In a commercial implementation of the MAP, it would probably be worth the effort to design a custom cell for the tag portion of each entry which would store a bit of the tag and perform the masked comparison of its contents with the address being translated. These cells could then be connected together using a wired-OR approach to generate the hit output for each entry, bringing the size of the GTLB much closer to that required to store the two words of data required for each entry

4.3 Using the M-Machine's Mechanisms to Implement Shared Memory[1]

Three threads on each MAP chip are primarily responsible for implementing shared memory: the event thread (thread 3 on cluster 0), which processes the events caused by remote

[1] This section presents a general description of how system software uses the MAP chip's hardware mechanisms to implement shared memory. Appendix A contains a detailed description of the shared-memory implementation used in this thesis.

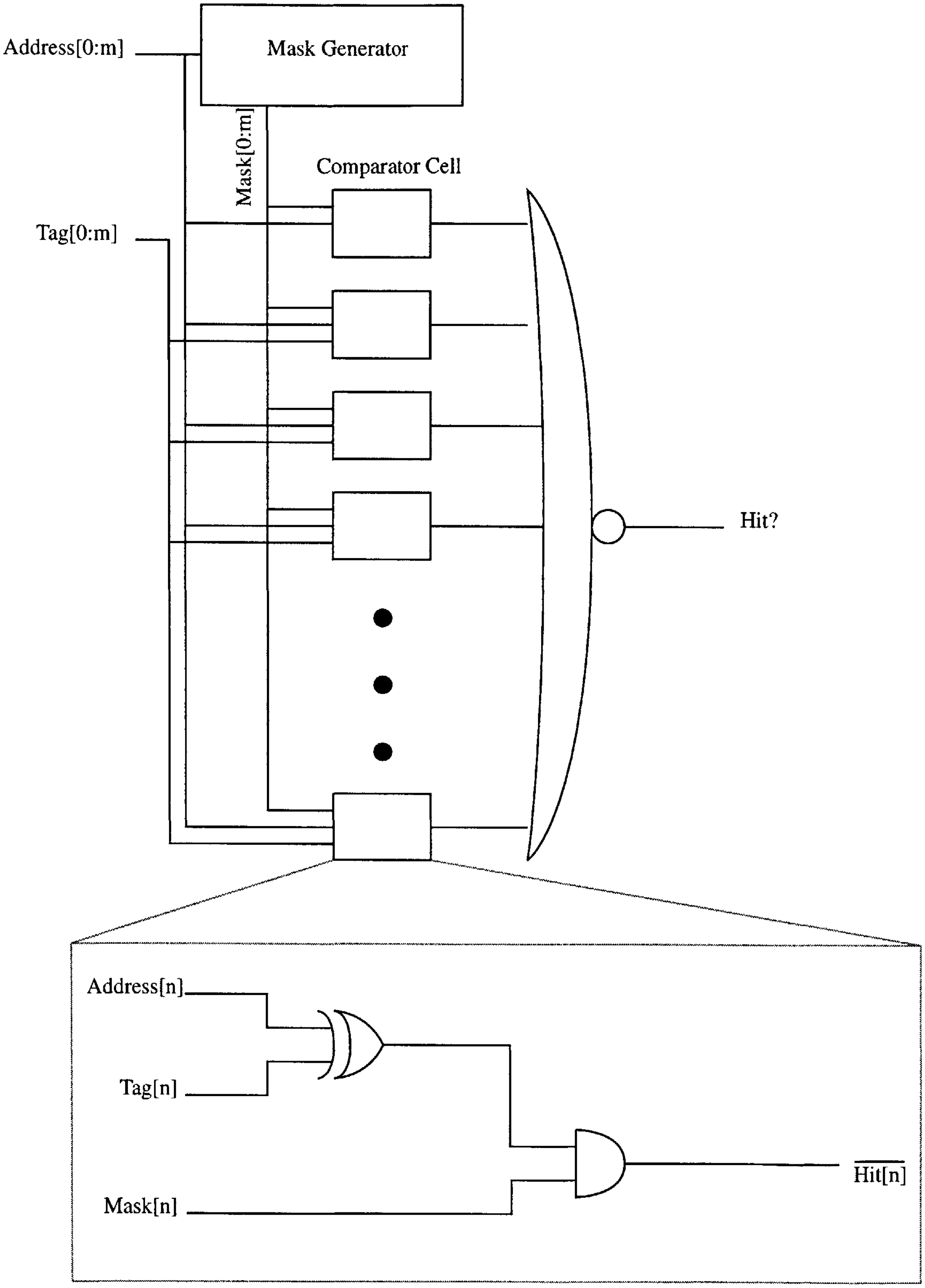


Figure 4.7: Masked Comparator

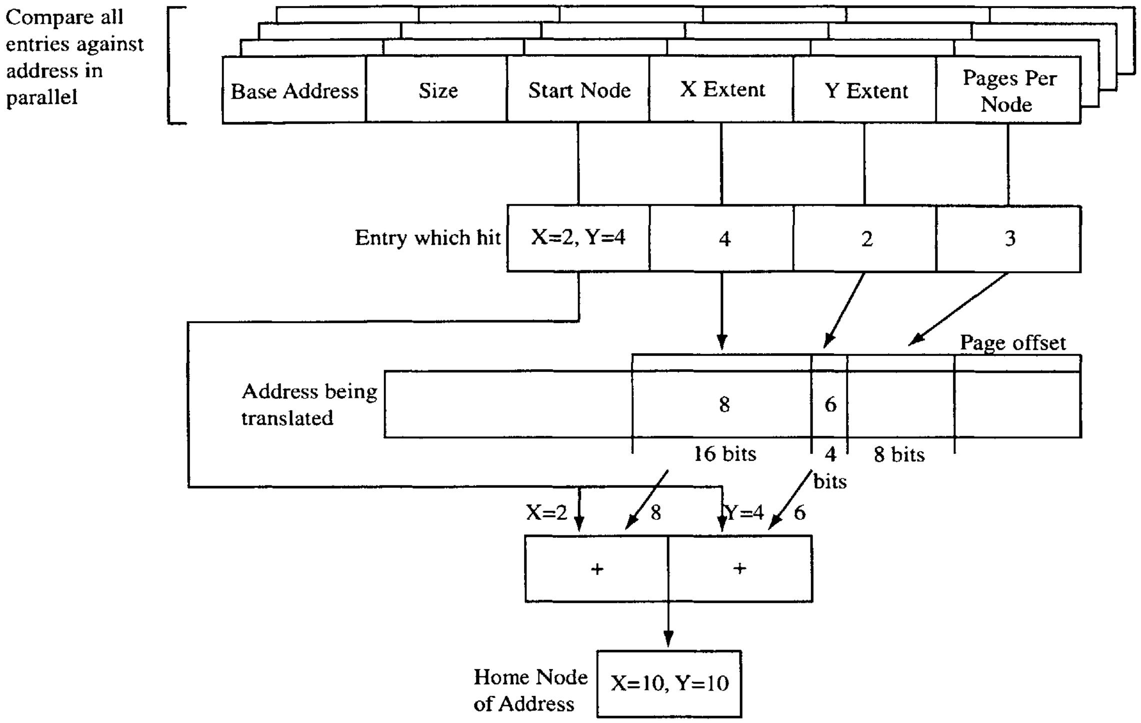


Figure 4.8: GTLB Translation

references and sends request messages for remote data; the priority 0 message handler thread (thread 4 on cluster 1), which handles request and invalidate messages, and sends reply messages; and the priority 1 message handler (thread 4 on cluster 1), which handles reply messages by installing requested lines and updating data structures. Two additional system thread slots (thread 3 on clusters 1 and 2) are used as *proxies* to enable the priority 0 message handler thread to send priority 0 messages in cases where three-node communication is required without the risk of deadlock that would occur if the handler attempted to send these messages directly

To implement a cached shared-memory protocol, two data structures are used by the shared-memory handlers on each node: the directory, and the pending operation structure. The directory keeps track of which nodes have copies of blocks of data that the node is the home node for, while the pending operation structure tracks the set of operations which are waiting for remote data before they can complete. The pending operation data structure is required to allow the hardware to discard operations which access remote data after the event queue entry for the operation has been created, allowing threads running on the node (such as the handlers which will resolve the remote reference) to continue execution while the reference is being resolved. The format and implementation of these data structures is discussed in detail in Appendix A.

Figure 4.9 shows the sequence of events required to resolve a simple remote reference (one which does not require the invalidation of any shared copies of the block containing the reference). Each step is annotated with (HW), (SW), or (HW/SW), to indicate whether the step is performed by the hardware, the software, or a combination of the two. A remote reference starts when a user program executes a memory operation which cannot be completed by the requesting node because the block status bits for the address do not allow it (the block is either invalid, or is read-only and a write operation is being performed).

Requesting Node	Home Node
1)User program executes reference to remote data (HW)	
2) Block status bit check determines that the reference cannot be completed (HW)	
3) Event system places block status miss in event queue (HW)	
4) Event handler begins, checks to see if request already sent for the block. (SW)	
5) Event handler creates pending operation record for request (SW)	
6) Event handler probes GTLB to find home node of address (HW/SW)	
7) Event handler sends P0 request message to home node (SW)	
	8) Request arrives in P0 queue (HW)
	9) Message handler checks block status of address to determine if eviction required (SW)
	10) Message handler adds requesting node to set of nodes that have a copy of the address (SW)
	11) Message handler changes status of its copy of the address if needed (SW/HW)
	11) Message handler sends P1 reply message with copy of the block containing the address (SW)
12) Reply arrives in P1 queue (HW)	
13) Message handler copies the block into memory (SW)	
14) Message handler sets block status bits of the block (HW)	
15) Message handler resolves all pending requests to the block. (SW/HW)	

Figure 4.9: Sequence of events in a remote reference

When the request reaches the memory system, the hardware checks the block status bits, and determines that the operation cannot proceed. This triggers the creation of an event queue entry which describes the operation and the reason why an event occurred. Once the event queue entry has been placed in the queue, the memory system hardware is free to execute other operations, and the event handler begins processing the event.

The event handler determines what type of event occurred, and branches to the correct handler, which extracts the address being referenced from the event queue entry, checks the pending operation structure to determine whether a request has already been sent for the block containing the address, and adds a record describing the operation to the list of operations waiting for the block. If a request for the block has already been sent, the event handler loops back to wait for the next event. If not, the event handler uses a **GPRB** instruction to determine the home node of the referenced address, and composes and sends a priority 0 request message for the address to the home node.

When the request message arrives at the home node of the address, the network hardware places it in the priority 0 message queue, and the priority 0 message handler thread begins resolving the message. First, the message handler executes a **GETCSTAT** operation to determine whether or not any copies of the block need to be evicted before the request can be satisfied. If the block is in the invalid state on the home node, then some other node has a writable copy of the line, which needs to be evicted before the requesting node can get a copy. If the block is in the read-only state on the home node, then one or more other nodes have read-only copies of the block, which need to be evicted before the requesting node can be given a writable copy of the block. Blocks which are in the writable or dirty state on their home nodes are not shared by any other node, so no eviction is required.

Requesting Node	Home Node	Sharing Node
1) Remote reference is detected, and request sent to home node. (HW/SW)		
	2) Message handler checks block status, determines invalidation required (HW)	
	3) Message handler records that invalidation of block has begun (SW)	
	4) Message handler signals proxy thread to send P0 invalidate messages to each sharing node (SW)	
		5) Invalidate message arrives in P0 queue (HW)
		6) Message handler sets block status of block to invalid (HW/SW)
		7) Message handler sends P1 data message to requesting node with copy of block. (SW)
		8) Message handler sends P1 ACK message to home node (SW)
9a) Data message arrives at P1 message queue (HW)	9b) ACK message arrives at P1 message queue.	
10a) Handler accesses pending operation structure to record arrival of data message (SW)	10b) Handler accesses directory to record arrival of ACK message (SW)	
11a) If all data messages have arrived, handler installs line and resolves pending operations (HW/SW)	11b) If all ACK messages have arrived, handler records that invalidation is done and that requesting node has copy of the block	

Figure 4.10: Remote memory access with invalidation

If no eviction is required, the message handler accesses the directory structure to add the requesting node to the list of nodes which have copies of the block, executes a **PUTCSTAT** operation to change the block status of its copy of the block to read-only if the request was for a read-only copy of the block, or invalid if the request was for a writable copy of the block. It then composes a reply message containing the contents of the block, which is sent to the requesting node on priority 1. Upon arrival at the requesting node, the reply message is copied into the priority 1 message queue by the hardware, and the priority 1 message handler begins resolving it.

The priority 1 message handler on the requesting node is responsible for copying the requested block into its memory and executing a **PUTCSTAT** operation to set the status of the block to its new value. It then iterates through all the operations which have been waiting for the block, performing each one in turn and using the configuration space to write the result of the operation directly into the destination register of the original operation. When all pending operations have been resolved, the reply handler loops back to wait for the next message.

If one or more node's copies of the line must be invalidated to complete the request, the procedure gets somewhat more complicated, as shown in Figure 4.10. Operations which are the same regardless of whether or not an invalidation occurs have been compressed in this figure to save space.

The first difference between this case and the previous example occurs when the request handler on the home node checks the block status of its copy of the line and determines that one or more invalidations must be performed. It then accesses the directory to record that the requested block is being invalidated. While the block is being invalidated, any other requests that arrive for it will be bounced back to the node which made the request, to be retried later. This prevents the block from getting into an inconsistent state because a copy of the block was given out while the block was being invalidated.

Once this has been done, the message handler signals a proxy thread to send invalidate messages to all of the nodes with copies of the block, and then loops back to wait for the next message. The proxy thread accesses the directory to find the list of nodes which have copies of the block, and sends an invalidate message to each one on priority 0.

When an invalidate message arrives at a sharing node, the message handler executes a **PUTCSTAT** operation to invalidate its copy of the block. It then sends a priority 1 data message with a copy of the block to the requesting node, and a priority 1 ACK message to the home node, and loops back to wait for the next message. Upon receipt of a data message, the priority 1 message handler on the requesting node accesses the pending operation data structure to determine how many data messages have arrived for the block, and compares that to the number of sharing nodes, which is contained in the data message.

If all data messages have been received, then all of the sharing nodes have invalidated their copies of the block, and the message handler is free to install the block and resolve operations waiting for the block, in the same way that it would have if no invalidation had been required. Otherwise, the message handler increments the number of data messages received, and waits for the next message to arrive.

The priority 1 message handler on the home node responds to ACK messages in much the same way, checking the directory structure to determine whether all the ACKs have been received and decrementing the number of pending ACKs if not. When the last ACK arrives, the message handler modifies the directory to record that the node which made the request is the only node which has a copy of the block, and that the invalidation is complete. If the home node was not the node which requested the block, the handler also copies the most recent copy of the block onto the home node and sets its block status to the appropriate value (invalid if the requesting node requested a writable copy of the block, read-only if the requesting node requested a read-only copy of the block). If the home node was the node which requested the block, the ACK handler does not install the line, as the data message handler is responsible for doing so.

Because the home node and the requesting node process the ACK and data messages independently, it is possible for the home node to process all of its ACK messages and conclude that the invalidation has completed before the requesting node has processed all of its data messages and installed the block it requested. This can lead to an inconsistent state in the protocol if the home node starts a second invalidation and the requesting node processes the invalidation message before it finishes installing the block, since the requesting node has a valid copy of the block, but the home node believes that the requesting node's copy of the block has been invalidated. To prevent this race condition, invalidation messages carry with them the state that they expect each sharing node's copy of the block to be in. During invalidation, the software checks the block status of the block to see if it matches the state expected by the invalidation message. If the status of the block does not match the expected status, the invalidation message is bounced back to the home node to be retried later, giving the sharing node time to process any pending data messages.

4.3.1 Varying the Shared-Memory Protocol

The framework described above for implementing shared memory on the M-Machine can easily be modified to support other protocols. Increasing the grain size of the protocol to fetch more data on each request simply requires modifying the system code to send multiple data messages for each request, one for each 8-word block[1]. The message handler which handles the data messages would also have to be modified to avoid changing the

[1] Because messages must be assembled in the sending thread's register file before they can be sent, it is not possible to send multiple blocks of data in a single message.

block status for any of the blocks being transferred until all had arrived. Modifying the protocol to support different grain sizes on different requests is possible, but more complicated, because this allows some nodes to have copies of only some of the blocks being transferred, making invalidation more difficult.

Implementing an uncached shared-memory protocol is also easy. Instead of transferring a block of data when a remote memory request is received, the handler on the home node simply performs the operation which caused the remote reference, and sends a message back to the requesting node with the result, allowing a handler on the requesting node to complete the operation. An update-based protocol (as opposed to the invalidation-based protocol presented earlier) can be implemented by setting the block status of all remote blocks that a node has copies of to read-only, and creating data structures which allow each node to track which other nodes have copies of its remote data. Whenever the node attempts to write any remote data, a block status event will occur because the data is in the read-only state, invoking an event handler which then updates the node's copy of the data and sends update messages to all other nodes with copies of the data. Alternately, a less-efficient protocol can be implemented by having the event handler send one update message to the home node of the address, which then sends update messages to each of the sharing nodes.

4.4 Discussion

One strong point of the M-Machine's implementation of shared memory is its support for storing remote data in the off-chip memory of a node in addition to the cache. This greatly increases the amount of remote data which can be stored on a node, reducing the frequency with which data must be fetched from another node. In addition, it moves the problem of evicting data to make room for incoming remote data off of the shared-memory handlers and onto the virtual-memory allocation system. In the MARS operating system which is used on the M-Machine, physical memory is allocated for a page by the virtual memory system as part of resolving the LTLB miss which occurs when the page is first accessed. Since LTLB misses take priority over block status events, the shared-memory handlers can assume that backing store exists for any remote address by the time the block status event handler is invoked. This simplifies the shared-memory handlers and reduces the remote memory access time, as the handlers never have to evict a block from the requesting node to make room for the block that they are requesting. If a node runs out of free physical pages, the virtual memory manager does have to evict remote pages to free space, but this can be done without impacting the shared-memory handlers by ensuring that pages with pending remote references are not evicted.

The MAP chip's implementation of block status bits does cause some complications during block installation and eviction. When evicting a block, it is necessary to first mark the block invalid, and then copy the contents of the block into the data message, in order to prevent the contents of the block from being modified after they are copied into the message. Similarly, during block installation, it is necessary to copy the contents of the block into the requesting node's memory and then change the status of the block to its new value.

Both of these operations require accessing a block in a manner not permitted by its block status value. To do this, the handlers generate the physical address of the block and access the block using the physical address, which bypasses the block status bits. This works, but, as will be seen in Chapter 6, adds a great deal of overhead to these operations which could be avoided if the MAP chip had implemented instructions which ignore the value of the block status bits.

Another weakness of the MAP chip is that it does not provide hardware support for varying the shared-memory protocol based on the address being referenced. This limits the ability of the programmer to use different shared-memory protocols to handle references to different regions of data, as the protocol handlers would have to perform a table lookup on the address of each remote memory reference to determine how references to the address should be handled. Adding a flag to each LTLB or GTLB entry which specified the shared-memory protocol to be used in on the data mapped by the entry would reduce this overhead, allowing greater flexibility and performance.

Finally, the M-Machine has difficulty implementing shared-memory protocols which preserve strict memory ordering, because the hardware can begin resolving requests to a block as soon as the block status bits of the block are set by the handler which installs the block, allowing requests to the block to be completed in hardware before the system software has completed all of the pending operations which have been waiting for the block to arrive. Implementing strictly-ordered shared memory protocols without modifying user programs requires that the system software halt all user threads whenever a block of remote memory arrives, wait until all in-flight memory operations have either completed or been added to the pending operation list by the event handler before installing the block, and then resolve all operations in the pending operation list which target the block before re-activating the user threads.

With help from the compiler, it is possible to implement strictly-ordered shared-memory protocols much more efficiently. The MAP chip implements a memory barrier instruction, known as **MBAR**, which causes the issuing thread to wait until all pending memory operations to the thread complete. If the compiler inserts an **MBAR** before each memory reference in a program, it can guarantee that all previous memory references have completed before the next memory reference executes, thus preserving strict memory ordering within each thread, albeit at the cost of preventing threads from tolerating latency by scheduling memory operations in advance of when their data is needed. This cost can be reduced somewhat by using memory disambiguation techniques to locate operations which are guaranteed not to reference the same address, and which therefore do not require the addition of barrier operations to maintain memory ordering.

Chapter 5

Evaluation

The previous chapter described the M-Machine's hardware mechanisms for shared memory, and showed how these mechanisms can be used to implement a variety of shared-memory protocols. This chapter describes the performance of a simple shared-memory protocol on the M-Machine, showing the impact of the MAP chip's mechanisms on both remote memory access time and the execution time of two applications. Microbenchmark results show that the MAP chip's GTLB and multithreading scheme provide similar improvements to the base remote access time, but that the multithreading scheme is more important to the performance of remote memory accesses which require invalidations. The program-level experiments show that the M-Machine achieves good speedups on shared-memory programs, including more than a factor of four speedup on an 8-node FFT benchmark.

5.1 Methodology

The experiments reported in this chapter and the next were run on the MSIM simulator, which simulates a version of the M-Machine in which each processor cluster contains a floating-point unit as well as the integer and memory units. The shared-memory handlers need floating-point units in each cluster because the M-Machine's network interface requires that all messages be assembled in the register files of the sending program before being sent. Some of the messages used in the cache-coherence protocol are too large to fit in the integer register files of the event or message handler threads, because two of the registers in these register files are used to interface with the event and network queues, and must therefore be assembled in the floating-point register file.

The use of MSIM for these experiments introduces some inaccuracy into the results, because MSIM does not model the flight time of messages through the network. The delay between the execution of a **SEND** operation and the injection of the message into the network is modeled correctly, as is the time required to copy a message into the message input queue of a node, but all messages have a flight time of one cycle in the network. As the simulations run for this thesis use a maximum of 8 nodes, and therefore have a longest path of 4 hops through the 2-D mesh network, this effect is not significant when compared to the time spent in the shared-memory handlers, though it would become more significant if larger numbers of processors were simulated.

The cache-coherence protocol described in Appendix A has been implemented and integrated with the MARS run-time environment for the M-Machine. Several versions of the shared-memory handlers used to implement the cache-coherence protocol were written, taking advantage of different combinations of the M-Machine's mechanisms in order to allow the performance impact of individual mechanisms to be evaluated. Virtual addresses are divided into a code space, which is replicated on each processor, and a data space, in which pages are allocated in round-robin fashion across the processors. This allocation scheme spreads the workload of supporting shared memory across the processors while still allowing the entire address space of the M-Machine to be described by two GTLB entries on each processor, one for the code space and one for the data space.

Two sets of experiments were run to evaluate the M-Machine's mechanisms for shared memory: a set of microbenchmarks which measured remote memory access times, and a set of program simulations, which show the impact of the mechanisms on program execution times. To generate fair microbenchmark numbers, assembly-language programs were written that performed the microbenchmark a large number of times, accessing a different block of memory each time, and the microbenchmark timings taken after the time per iteration had converged. This simulates the expected steady-state case when the shared-memory handlers are being used to implement real programs: all of the code and data used by the shared-memory handlers is present in the caches, but the remote address being referenced is not in the cache on the home node.

Two programs were simulated to assess the impact of the M-Machine's mechanisms on program performance: a fast-Fourier transform (FFT), and a multigrid solver. The FFT program was chosen because of its importance in signal-processing applications, while the multigrid solver is typical of scientific applications. The C-language source code for these applications was provided by the Alewife group at M.I.T.[4], and compiled on the mmcc compiler for the M-Machine.

Porting these applications to the M-Machine required that a number of assembly language routines be written to perform tasks such as forking parallel processes, barriers, and fetch-and-add. These routines were then linked in with the compiled C programs to generate an executable. In addition, a number of memory barriers (MBAR instructions) had to be added to the programs to enforce correct memory ordering, as the compiler was written with the assumption of a strictly-ordered memory model within each thread. This step proved far more difficult than had been anticipated, partially due to the fact that MARS does not allow programs to allocate memory locally on a node. Instead, all memory allocated by programs, including stack space, is part of the global data space, which is distributed across the nodes. This made it much more difficult to predict where it would be necessary to add memory barriers to the program. The difficulty encountered in modifying the source code to provide correct memory ordering argues that compiler support to automatically add memory ordering instructions to programs is necessary for shared-memory protocols that do not provide strict memory ordering to be successful.

5.2 Microbenchmark Results

Figure 5.1 shows the time to perform a simple remote memory access as a function of the set of hardware mechanisms used by the protocol. The total remote access time has been divided into three categories to show the time spent in the event, request, and reply handlers. For these measurements, the time spent in the event handler includes the total time from the execution of the remote memory reference until the start of the request handler on the home node, including the network delay to send the request message. Similarly, the request handler time measures the delay from the start of the request handler to the start of the reply handler on the requesting node, and the reply handler time measures the delay from the start of the reply handler to the execution of an instruction which has been waiting for the result of the original remote reference, including the time to write the result of the original remote reference into the requesting thread's register file via the configuration space.

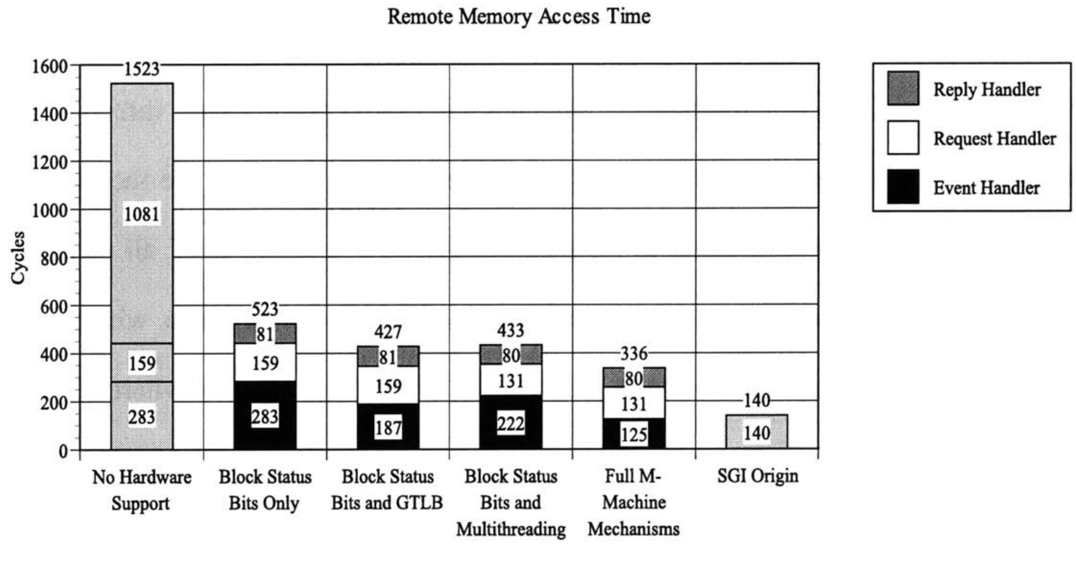


Figure 5.1: Remote Memory Access Times

The leftmost column of the graph shows an estimate of the M-Machine's remote access time (1523 cycles) if none of the hardware mechanisms were used, requiring the handlers to transfer an entire page of data between nodes to resolve each remote memory reference, in addition to the work required to resolve a remote memory reference using only the block status bits. This estimate is based on the performance of a block transfer handler written by Whay Lee, and was generated by adding the expected transfer time for a 4KB page to the remote access time shown in the second column, in which the block status bits were the only hardware mechanisms used, yielding a remote access time of 523 cycles.

Making use of the GTLB in addition to the block status bits reduces the access time from 523 cycles to 427, an 18% improvement, while using the MAP chip's multithreading scheme and the block status bits gives a remote access time of 433 cycles, a 17% improvement over using just the block status bits. Taking advantage of the GTLB, the multithreading scheme, and the block status bits gives a remote access time of 336 cycles with all the M-Machine's hardware mechanisms in use, a speedup of 36% over the block status bits alone. The 140-cycle remote access time of an 8-node SGI Origin, as reported in [20], is shown in the rightmost column as an example of the performance achieved by a contemporary full-hardware shared-memory system.

5.2.1 Remote Access Time Breakdown

Figure 5.2 shows the steps involved in resolving a remote memory reference when all of the M-Machine's hardware mechanisms are in use. On cycle 0, a user program issues a memory operation which references data which is not present on the local node. By cycle 10, the event system has checked the block status bits of the referenced address, determined that the data is not present on the node, and started the event thread by placing an event record describing the reference in the event queue. The next 23 cycles are spent extracting the event type from the header word of the event record, indexing into a software jump table to determine the address of the event handler which will be used to resolve the event, and jumping to the appropriate handler.

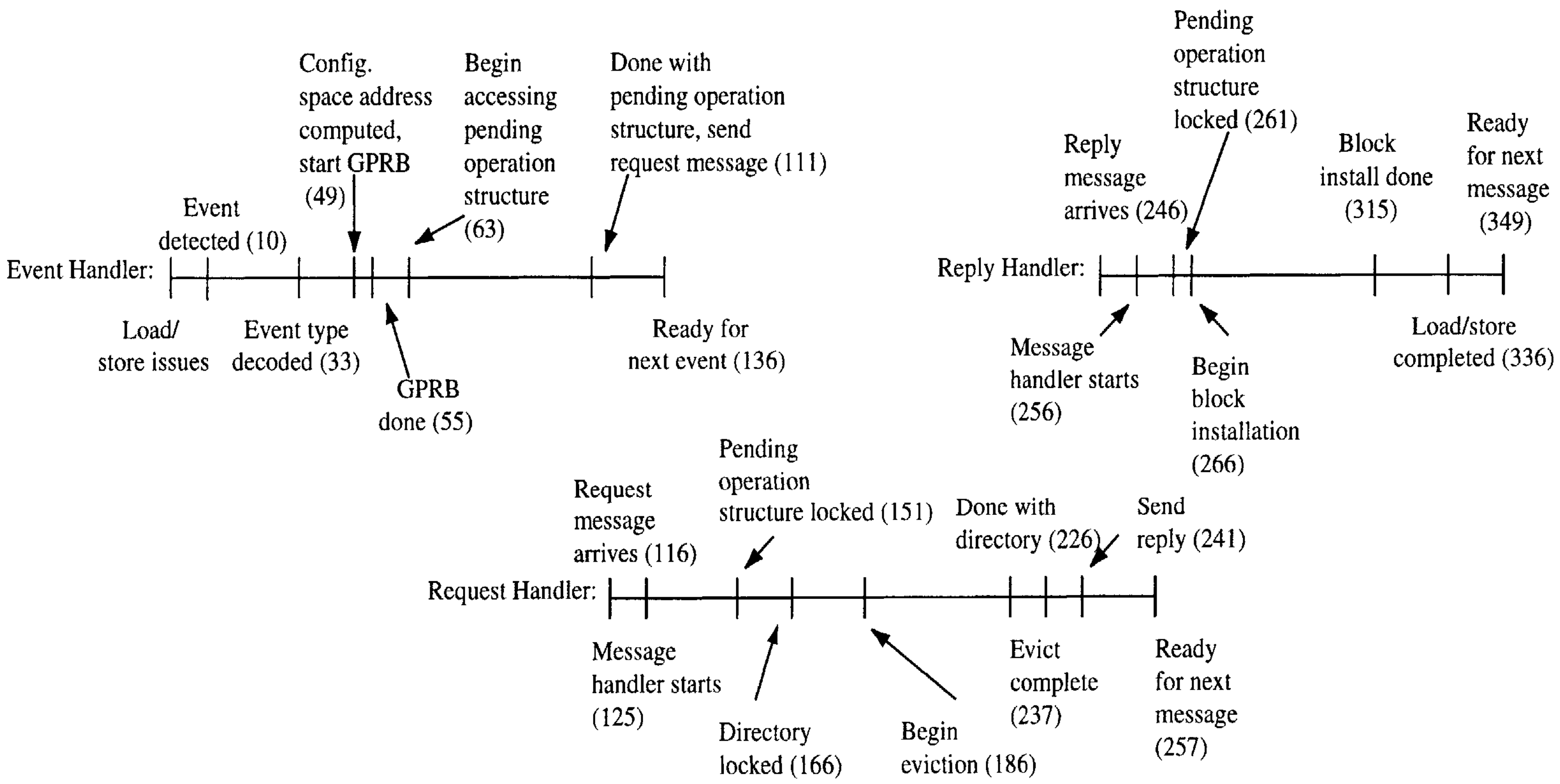

Event Handler:
Event detected (10)
Config. space address computed, start GPRB (49)
Begin accessing pending operation structure (63)
Done with pending operation structure, send request message (111)
Load/ store issues
Event type decoded (33)
GPRB done (55)
Ready for next event (136)
Reply Handler:
Reply message arrives (246)
Pending operation structure locked (261)
Block install done (315)
Ready for next message (349)
Message handler starts (256)
Begin block installation (266)
Load/store completed (336)
Request Handler:
Request message arrives (116)
Pending operation structure locked (151)
Done with directory (226)
Send reply (241)
Message handler starts (125)
Directory locked (166)
Begin eviction (186)
Evict complete (237)
Ready for next message (257)

Figure 5.2: Remote Memory Request Timeline

Once started, the event handler generates the configuration space address which will be used to resolve the operation that caused the event, based on the information contained in the event record. It completes this by cycle 49 and then executes a GPRB instruction to find the home node of the referenced address, which returns on cycle 55. On cycle 63, the event handler begins accessing the pending operation structure to determine if a request has already been sent for the block containing the referenced address and to enqueue a pending operation record for the operation. By cycle 111, the pending operation record has been created, and the event handler sends a request message to the home node. The event handler then spends 25 cycles prefetching the physical address of the referenced virtual address into the cache to accelerate the process of installing the block when the reply message arrives. On cycle 136, the event handler is ready to handle the next event.

Meanwhile, the request message has arrived on the home node by cycle 116, and the message handler thread has jumped to the correct message handler by cycle 125. By cycle 151, the request handler has locked the entry in the home node's pending operation structure corresponding to the page being referenced, to prevent any other handlers from changing the state of the page on the home node while the request handler is running. On cycle 166, the request handler locks the appropriate entry in the directory. The request handler then checks the block status of the referenced address on the home node to determine if it is necessary to invalidate some other node's copy of the block, and begins evicting its copy of the block by cycle 186. Eviction of the block is interleaved with accessing the directory to add the requesting node to the list of nodes which have a copy of the block, causing the eviction to complete on cycle 237, after the handler is done accessing the directory on cycle 226. On cycle 241, the request handler sends the reply message containing the block back to the requesting node, and is ready to handle the next message by cycle 257.

The reply message arrives at the requesting node by cycle 246, and the reply handler begins execution on cycle 256. By cycle 261, the reply handler has locked the pending operation data structure, and installation of the block begins on cycle 266. Installation of the block is complete on cycle 315, and the handler begins completing all of the operations which have been waiting for the block. On cycle 336, the handler completes the first (and only, in this case) pending operation, and the handler thread is ready for the next message on cycle 349. Because the configuration space allows the reply handler to write the return value of each of the pending operations directly into the destination register of the original operation, operations which depend on the results of these operations may proceed as soon as the pending operations have been completed by the reply handler.

One difference between the shared-memory protocol implemented on the M-Machine and other software shared-memory protocols is that each of the handlers performs all required data structure modifications before sending any required messages, rather than sending the messages first to reduce latency. This is done to prevent deadlock in the shared-memory system. Each of the data structures used by the shared-memory system is divided into individual records, and the software handlers are required to acquire locks on these records before accessing them to prevent multiple handlers from modifying the same record simultaneously. This allows the shared-memory system to exploit parallelism across the different handler threads running on a node. However, a handler may not send a message while holding the lock on a software structure, as this creates a potential deadlock situation involving the event handler on the requesting node, the request handler on the home node, and the reply handler on the requesting node, as shown in Figure 5.3.

To prevent this deadlock from occurring, the handlers in the shared-memory protocol are not allowed to send any messages while they have locks on any of the data structures. Since the event and request handlers need to access their data structures to determine how

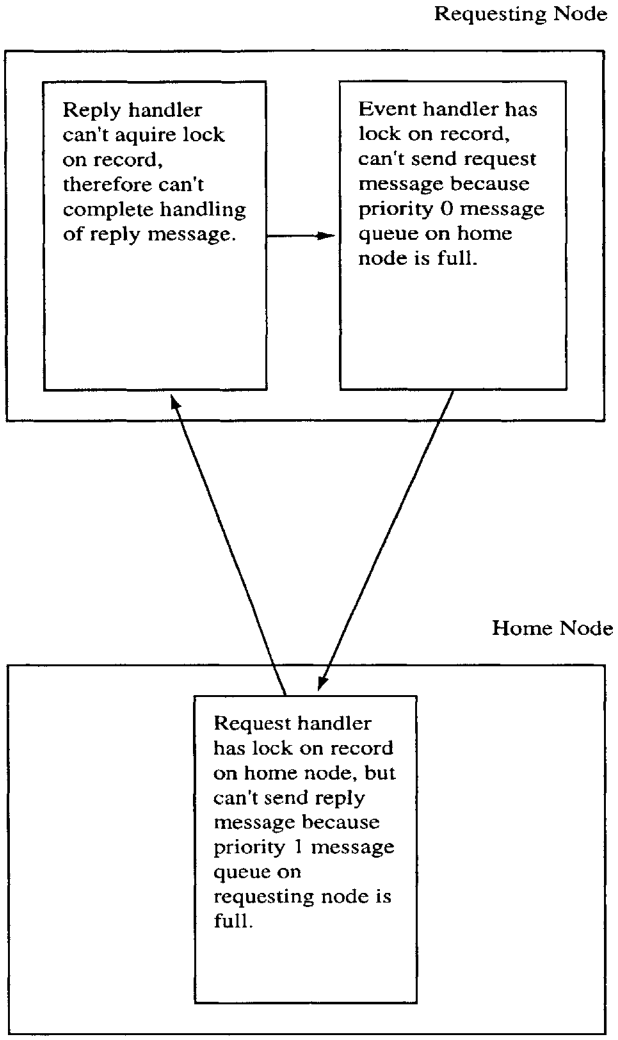

Requesting Node
Reply handler can't aquire lock on record, therefore can't complete handling of reply message.
Event handler has lock on record, can't send request message because priority 0 message queue on home node is full.
Home Node
Request handler has lock on record on home node, but can't send reply message because priority 1 message queue on requesting node is full.

Figure 5.3: Potential Deadlock Situation

to respond to a remote memory reference, it is not possible for them to send their messages before acquiring the locks that they need on the data structures. Therefore, it is necessary for the handlers to acquire locks on the data structures, perform all necessary modifications to the structures, release the locks, and then send any messages required by the protocol. It would be possible for the handlers to release their locks in order to send their messages and then re-acquire the locks to complete the data structure modifications after the messages had been sent, but this would complicate the protocol by introducing the possibility that some other handler could modify a record between the time that a handler released its locks and the time that it re-acquired the locks.

5.2.2 Impact of Mechanisms on Remote Access Time

Figure 5.1 showed that the MAP chip's mechanisms significantly improve the M-Machine's remote memory access time. To better understand where these improvements come from, this section will discuss how remote memory references are resolved by the handlers when some or all of the MAP chip's mechanisms are not used.

Figure 5.4 shows a timeline of the resolution of a remote memory request when the block status bits and the multithreading scheme are used, and the GTLB is not. Detecting that a remote access has occurred and starting the appropriate event handler takes 34 cycles. Translation of the referenced address begins on cycle 50, and takes 101 cycles, completing on cycle 151.

Once the home node of the referenced address has been determined, the event handler enqueues the pending operation record for the operation in the pending operation structure, and sends out a request message on cycle 208. It then takes the event handler 26 more cycles to complete execution and be ready for the next event, as compared to the 21 cycles required for the event handler when all of the mechanisms are in use. The additional 5 cycles of delay are incurred because of the need to restore the contents of a few integer registers which had to be saved away in the floating-point unit due to the increased register pressure in this version of the handler.

The request and reply handlers are unchanged in this version of the protocol. The request message arrives at the home node on cycle 213, 97 cycles after the request message arrives in the base version of the protocol. The reply message is sent to the requesting node on cycle 338, also 97 cycles later than in the base protocol, and this 97-cycle gap is

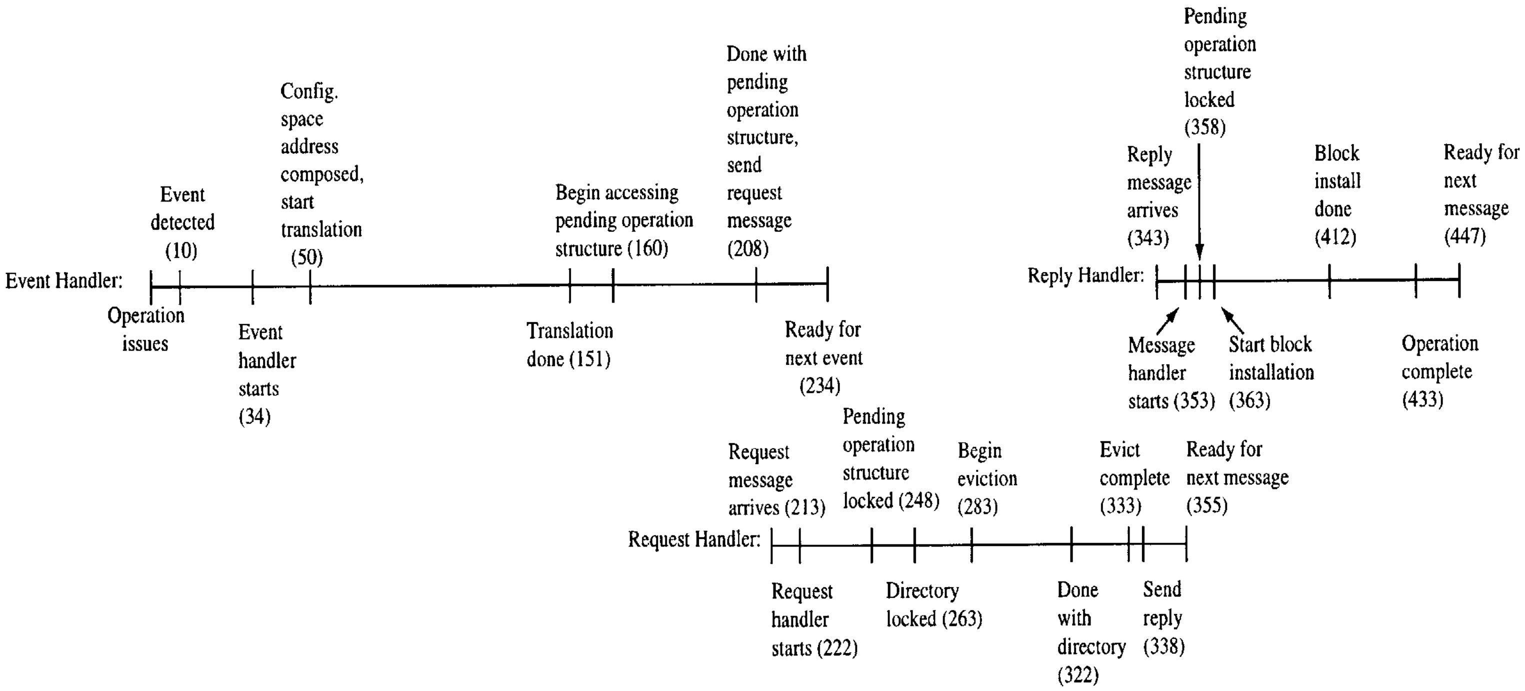

Event Handler:
Event detected (10)
Config. space address composed, start translation (50)
Begin accessing pending operation structure (160)
Done with pending operation structure, send request message (208)
Operation issues
Event handler starts (34)
Translation done (151)
Ready for next event (234)
Request Handler:
Request message arrives (213)
Pending operation structure locked (248)
Begin eviction (283)
Evict complete (333)
Ready for next message (355)
Request handler starts (222)
Directory locked (263)
Done with directory (322)
Send reply (338)
Reply Handler:
Pending operation structure locked (358)
Reply message arrives (343)
Block install done (412)
Ready for next message (447)
Message handler starts (353)
Start block installation (363)
Operation complete (433)

Figure 5.4: Remote Memory Request Without GTLB

maintained through the completion of the operation which caused the remote miss on cycle 433.

One significant difference between this version of the protocol and the version which uses all of the mechanisms is the distribution of time across the handlers in the protocol. In the base case, the event handler thread slot is occupied for 126 cycles per request (136 minus the 10 cycles required to create the event queue entry), the request handler occupies its thread slot for 141 cycles, and the reply handler executes for 103 cycles, allowing a sequence of requests for data with the same home node to be resolved at a rate of one every 141 cycles, limited by the execution time of the request handler. When the GTLB is not used, the event handler's execution time increases to 224 cycles, while the execution times of the request and reply handlers remain relatively unchanged at 142 cycles and 104 cycles, respectively.

Because the additional latency and occupancy which results from not using the GTLB is not evenly distributed across the different handlers, the program-level impact of not using the GTLB may vary depending on the access pattern of the program. If one node in the machine sustains significantly more remote memory references than the others, as might occur in a master-slave parallelization in which the master thread needs to access all of the data structures used by the slave threads, then the time to process all of the events on the requesting node will be the bottleneck, assuming the home nodes of the requested data are reasonably evenly distributed. In such a case, performing address translation in software instead of hardware will significantly reduce performance, due to the increase in execution time of the event handler. If, instead, the program contains a great deal of contention among the nodes for a small number of addresses, then there will be little difference in overall performance as a result of using hardware or software address translation, as remote memory bandwidth will be determined by the execution time of the request handlers on the home nodes of the referenced addresses, which do not change significantly when the method of address translation changes.

If the shared-memory handlers make use of hardware address translation but not the MAP chip's multithreading scheme, the increase in remote memory latency is much more evenly distributed across the handlers, as shown in Figure 5.5. This version of the shared-memory protocol simulates the execution of software shared memory on a non-multithreaded processor with a fast interrupt mechanism by saving the contents of the register file on the stack at the start of each handler and restoring them at the end, as if the hardware were interrupting the execution of a user program in order to use the processing resources to execute shared-memory handlers. Other than this, the handlers are identical to those used when all of the mechanisms are in use, so the execution times of the handlers should be almost identical to the base case once the thread-swapping is complete.

In the event handler, this thread-swapping increases the delay from the execution of a remote reference to the start of the event handler for that reference from 33 cycles to 66 cycles. This 33-cycle offset causes the request message to arrive at the home node 33 cycles later than in the base protocol. Simulating swapping the user thread back in at the end of the event handler increases the delay between the message send and the point at which the handler completes by 30 cycles, due to the need to wait for the loads to complete before execution can resume, making the total execution time of the event handler 199 cycles.

Similarly, the time from the arrival of the request message to the start of the message handler increases by 29 cycles, and the reply message is sent out 62 cycles later than the reply message in the base protocol. Another 28 cycles of delay accumulate during the invocation of the reply handler and the reply handler takes one more cycle to execute than in the base case, leading to a total increase in remote access time of 91 cycles.

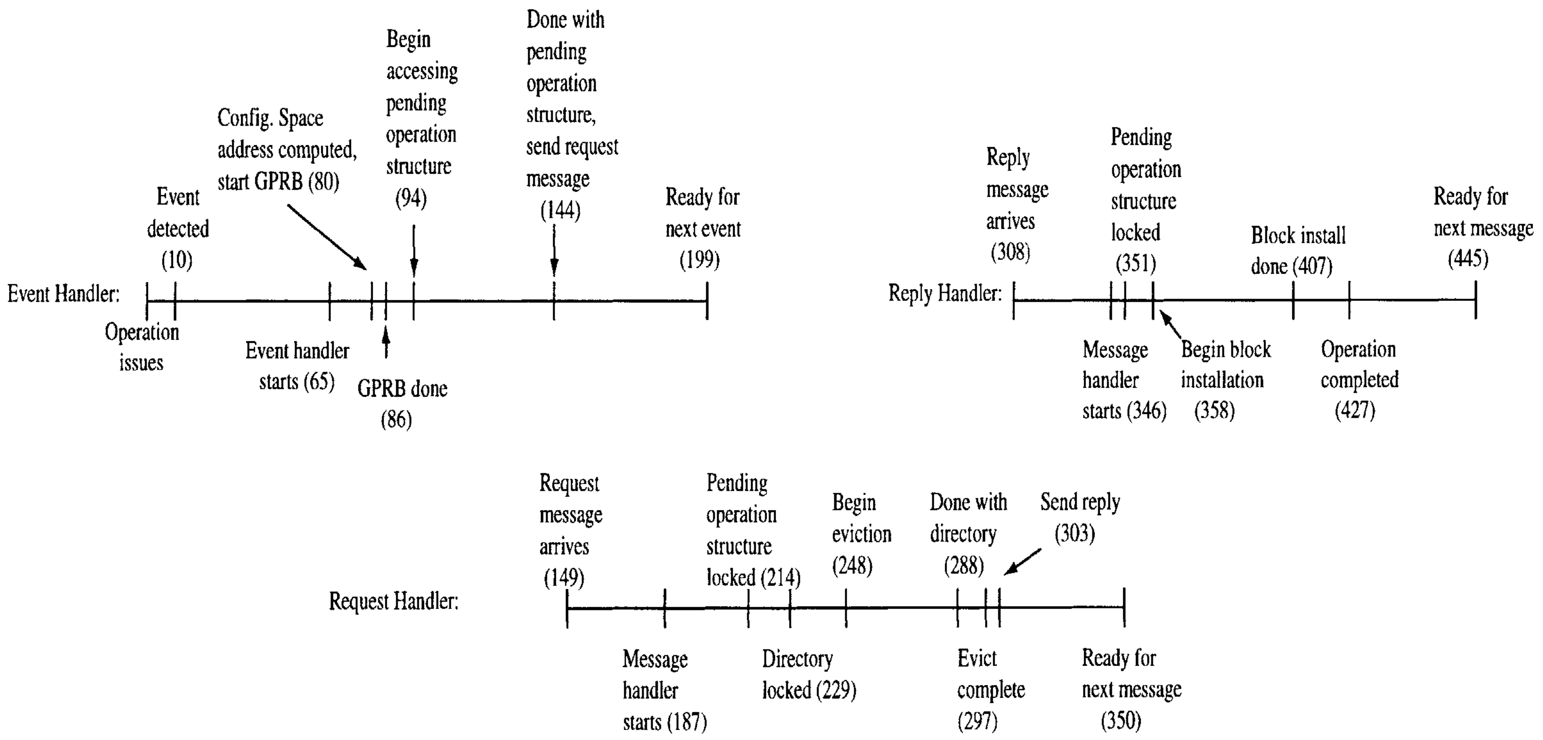

Event Handler:
Event detected (10)
Config. Space address computed, start GPRB (80)
Begin accessing pending operation structure (94)
Done with pending operation structure, send request message (144)
Ready for next event (199)
Operation issues
Event handler starts (65)
GPRB done (86)
Reply Handler:
Reply message arrives (308)
Pending operation structure locked (351)
Block install done (407)
Ready for next message (445)
Message handler starts (346)
Begin block installation (358)
Operation completed (427)
Request Handler:
Request message arrives (149)
Pending operation structure locked (214)
Begin eviction (248)
Done with directory (288)
Send reply (303)
Message handler starts (187)
Directory locked (229)
Evict complete (297)
Ready for next message (350)

Figure 5.5: Remote Memory Reference Without Multithreading

In this version of the protocol, all three handlers see similar increases in their execution times as compared to the base case. The event handler takes 189 cycles, an increase of 63 cycles. The request handler's execution time increases by 60 cycles, to a total of 201, while the reply handler executes in 137 cycles, 34 cycles more than the base case. This relatively even distribution of the change in latency suggests that the performance improvement which results from the use of multithreading in the shared-memory system should be relatively independent of the access patterns of application programs, since the performance improvement is distributed across all three handler threads. In general, remote memory bandwidth will still be limited by the execution time of the request handler, although less imbalance in access patterns will be required to make the event handler thread the bottleneck, as the percent difference between its execution time and that of the request handler has shrunk.

If both the M-Machine's multithreading scheme and the GTLB are disabled, leaving the block status bits as the only mechanism in use, the change in remote memory latency is approximately the sum of the changes resulting from disabling the two mechanisms independently, as shown in Figure 5.6. The total increase in remote memory latency as compared to the base case is 187 cycles, one cycle less than the sum of the differences which result when each of the mechanisms are disabled independently.

The execution times of the event, request, and reply handlers in this version are 281 cycles, 201 cycles, and 164 cycles, respectively, making them more imbalanced than the base version of the protocol, but less imbalanced than the version which performed address translation in software but made use of the MAP chip's multithreading scheme. In general, the execution time of the event handler should be the bottleneck on remote-memory bandwidth in this version, although load imbalances could cause the request handler time to be more significant if there is contention for a few memory addresses.

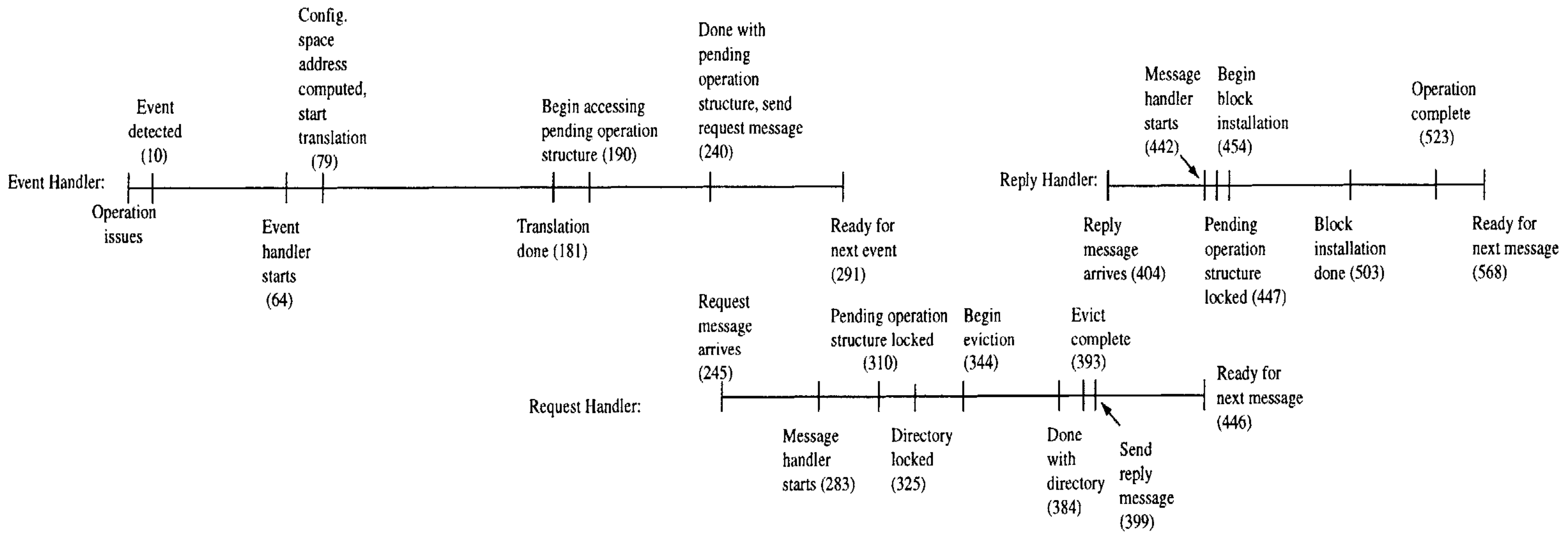

Event Handler:
Event detected (10)
Operation issues
Event handler starts (64)
Config. space address computed, start translation (79)
Translation done (181)
Begin accessing pending operation structure (190)
Done with pending operation structure, send request message (240)
Ready for next event (291)
Request Handler:
Request message arrives (245)
Message handler starts (283)
Pending operation structure locked (310)
Directory locked (325)
Begin eviction (344)
Done with directory (384)
Evict complete (393)
Send reply message (399)
Ready for next message (446)
Reply Handler:
Reply message arrives (404)
Message handler starts (442)
Pending operation structure locked (447)
Begin block installation (454)
Block installation done (503)
Operation complete (523)
Ready for next message (568)

Figure 5.6: Remote Memory Request Using Block Status Bits Only

5.2.3 Remote Memory Access Times With Invalidation

If it is necessary to invalidate one or more copies of a block, the protocol becomes more complex, as shown in Figure 5.7, which illustrates the process of invalidating a block which is shared by three nodes other than the home node, using all of the M-Machine's mechanisms. The event handler cannot tell whether a request will require invalidation, and therefore handles the remote memory reference in exactly the same manner as a reference which does not require invalidation, although the execution time of the handler varies somewhat from the simple remote reference case due to interactions with the rest of the MAP chip.

When the request message arrives on cycle 110, the behavior of the handlers begins to diverge from the simple remote reference. Early in its execution, the request handler checks the block status of the requesting block on the home node, and determines that one or more invalidations are required. It then probes the directory to find the list of nodes which have copies of the block, and passes this information to the evict proxy thread, which begins execution on cycle 216. Once the request handler has signalled the evict proxy thread to handle the invalidations, it terminates, and the message handler thread slot is ready to handle the next request on cycle 232.

By cycle 258, the evict proxy thread has composed the invalidate message and extracted the first word of the list of sharing nodes from the directory, allowing it to send the first invalidate message to a sharing node. The second and third invalidate messages are sent out on cycles 276 and 293, respectively. Most of the delay between messages is due to the time that the network unit takes to copy the contents of the message out of the evict proxy thread's register file and into the network output queue, although there is some time spent extracting the next node ID from the directory and setting up for the next message.

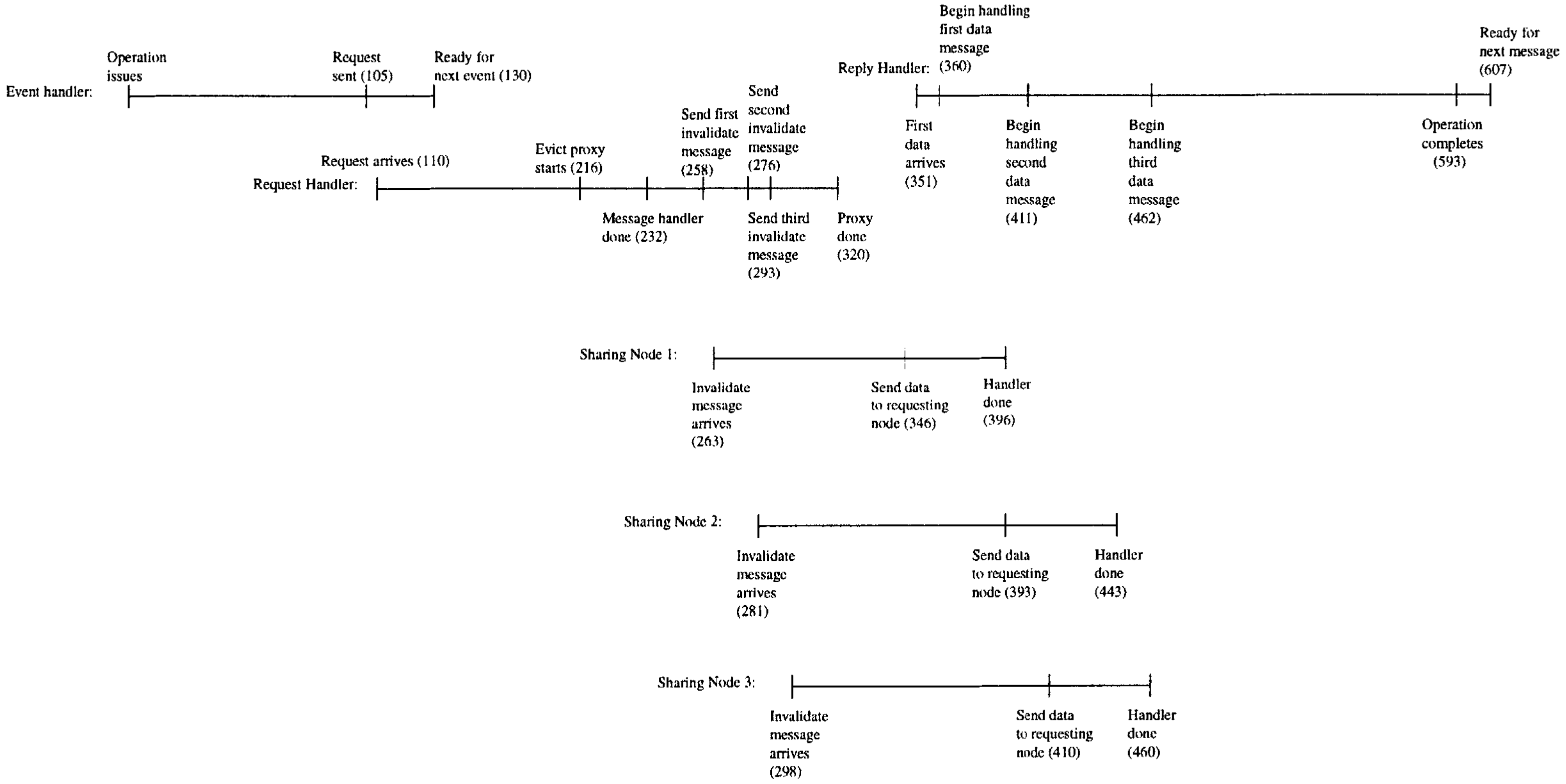

Event handler:
Operation issues
Request sent (105)
Ready for next event (130)
Request Handler:
Request arrives (110)
Evict proxy starts (216)
Message handler done (232)
Send first invalidate message (258)
Send second invalidate message (276)
Send third invalidate message (293)
Proxy done (320)
Reply Handler:
Begin handling first data message (360)
First data arrives (351)
Begin handling second data message (411)
Begin handling third data message (462)
Operation completes (593)
Ready for next message (607)
Sharing Node 1:
Invalidate message arrives (263)
Send data to requesting node (346)
Handler done (396)
Sharing Node 2:
Invalidate message arrives (281)
Send data to requesting node (393)
Handler done (443)
Sharing Node 3:
Invalidate message arrives (298)
Send data to requesting node (410)
Handler done (460)

Figure 5.7: Invalidation Timeline

The invalidate messages arrive at the sharing nodes on cycles, 263, 281, and 298 on network priority 0. The invalidation handler on the first sharing node takes 83 cycles to process its invalidation message, evict the specified block, and send a data message back to the requesting node, which arrives at cycle 351. The second and third sharing node's invalidate handlers take 112 cycles to send their data messages, due to back-pressure from the network input queue on the requesting node.

Each of the first two data messages to arrive at the requesting node take 51 cycles to process, which includes the time to parse the message, access the pending operation structure to determine if the required number of data messages have arrived, and record that an additional message has arrived. The third data message takes 145 cycles to process, as the message handler must install the block and resolve the operation which requested the block once it has determined that all of the data messages have arrived.

Handling the data messages on the requesting node is the bottleneck of the invalidation process. Sending each invalidation message takes 17-18 cycles, although this increases slightly every fourth message, as the directory stores the node ID's of up to four sharing nodes in a word of data, requiring that a new word of data be fetched from the directory every four messages. In contrast, resolving data messages other than the last one takes 51 cycles on the requesting node, limiting the rate at which invalidations can be processed and increasing the occupancy of the invalidate handlers running on the sharing nodes due to network back-pressure.

5.2.4 Effect of Mechanisms on invalidation Time

Figure 5.8 shows the M-Machine's remote access time as a function of the set of mechanisms used by the protocol and the number of invalidations required to complete the reference. With the exception of the jump from 0 to 1 invalidation, the slope of each line remains relatively constant, as the incremental cost of each invalidation is determined by the time to process an additional data message on the requesting node.

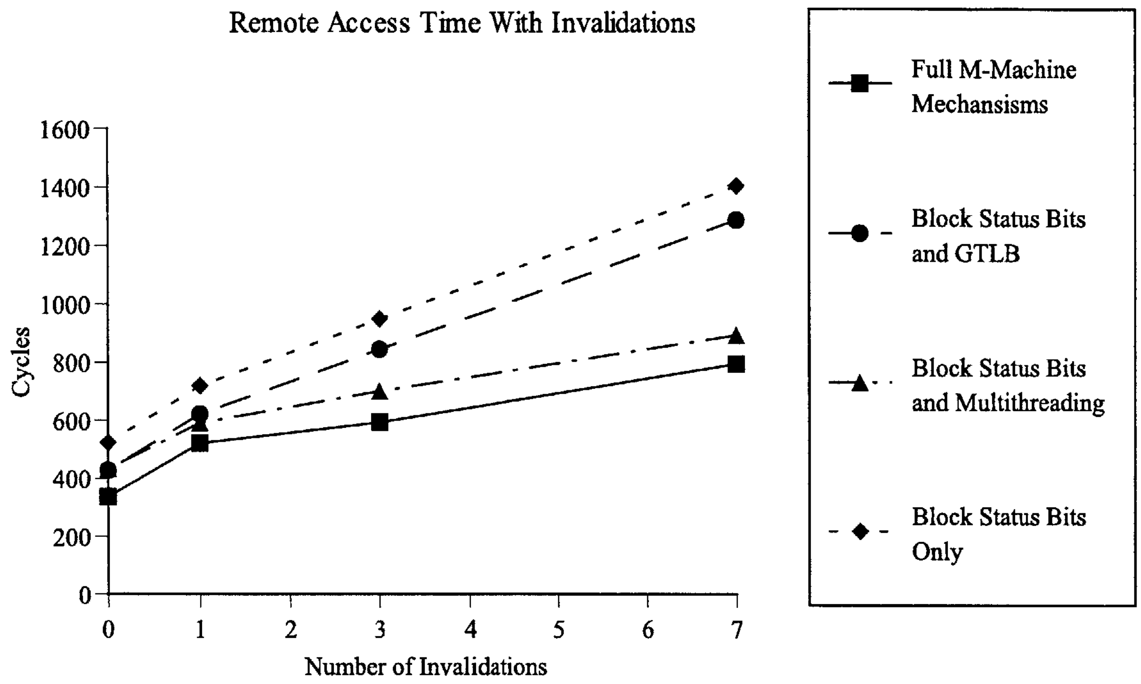


Figure 5.8: Remote Access Times With Invalidations

One interesting feature of this graph is that the effect of not using the GTLB is constant with respect to the number of invalidations required, while the performance impact of not using the MAP chip's multithreading scheme increases linearly with the number of invalidations. This occurs because address translation is done in the event handler, which only executes once per request regardless of the number of invalidations required to resolve the request. In contrast, executing the shared-memory protocol on a single-threaded processor adds a context switch overhead to the execution time of each handler on the processor, which increases the time to process each data message on the requesting node. Because of this effect, the protocol which makes use of the multithreading scheme and the block status bits achieves better performance than the protocol which makes use of just the GTLB and the block status bits when invalidations are required, although the protocol which uses the GTLB and the block status bits has better performance for simple remote memory requests.

5.3 Program Results

To understand the impact of the M-Machine's mechanisms for shared memory, two programs have been simulated: a 1,024-point FFT computation, and an 8x8x8 multigrid computation. The times reported here are for the parallel core of each application, and do not include the execution time of data placement code on each node. For the program sizes run for this thesis, FFT and Multigrid display substantially different remote memory access patterns, showing the performance of the M-Machine as different characteristics of the shared-memory system become the bottleneck on performance. In FFT, the remote memory references are reasonably well distributed across the nodes, and overall utilization of the shared-memory handler threads is relatively low. This leads to good overall performance and relatively low sensitivity to changes in the shared-memory handlers. The multigrid benchmark, on the other hand, displays a very uneven remote access pattern, in which the majority of the remote accesses reference one node's data. This leads to high utilization of the request handler thread on that node, lowering overall performance and making the program much more sensitive to changes in the shared-memory handlers.

5.3.1 FFT

As shown in Figure 5.9, FFT performs well on the M-Machine, achieving a speedup of 4.2 with all of the mechanisms for shared memory in use. The execution time breakdown presented in Figure 5.10 shows that processor utilization is good, with the average processor spending 38% of its time executing instructions and only 19% waiting for data to return from memory. The greatest limitation on the performance of this program is synchronization delay imposed by barriers, which consume 34% of the execution time on average. This barrier delay is partially due to the fact that barriers are implemented using shared-memory references instead of an optimized message-passing routine, but over 75% of the barrier delay comes from the variance in arrival times at the barriers which results from load imbalance, indicating that the performance impact of replacing the shared-memory barrier with a message-based implementation would be small.

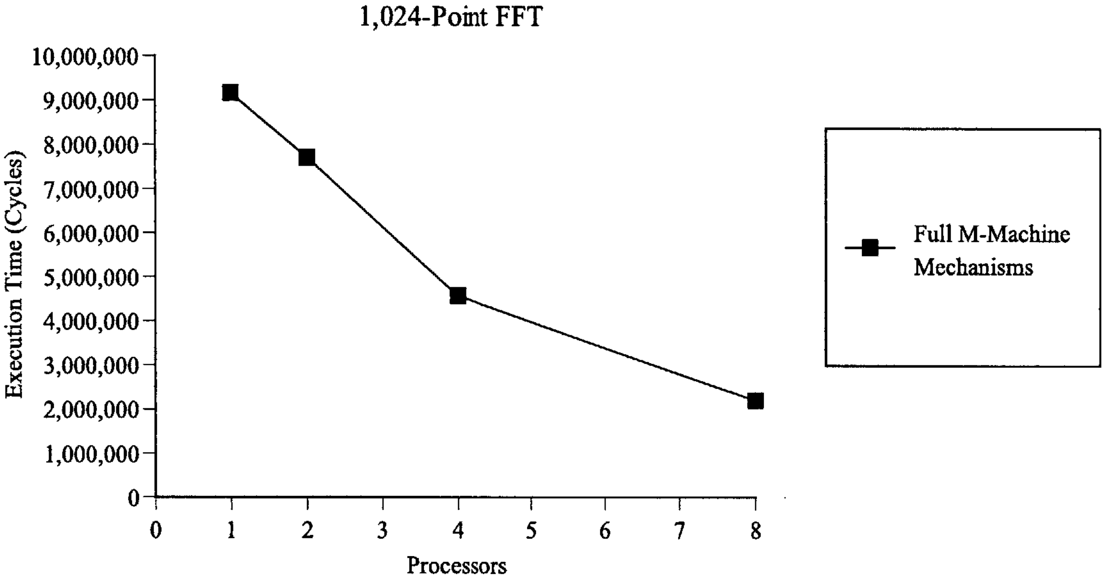


Figure 5.9: 1,024-Point FFT Execution Time

Figure 5.11 shows the occupancy rates of each of the shared-memory handlers during the execution of this program. Occupancy of all the handler threads is low, arguing that latency rather than occupancy is the limiting factor on shared-memory performance for this application. On all but one of the processors, the occupancy of the request handler thread is greater than either of the other two threads, due to the fact that the request handler handles both remote memory requests and invalidation requests and the longer execution times of the handlers than run in the request handler slot.

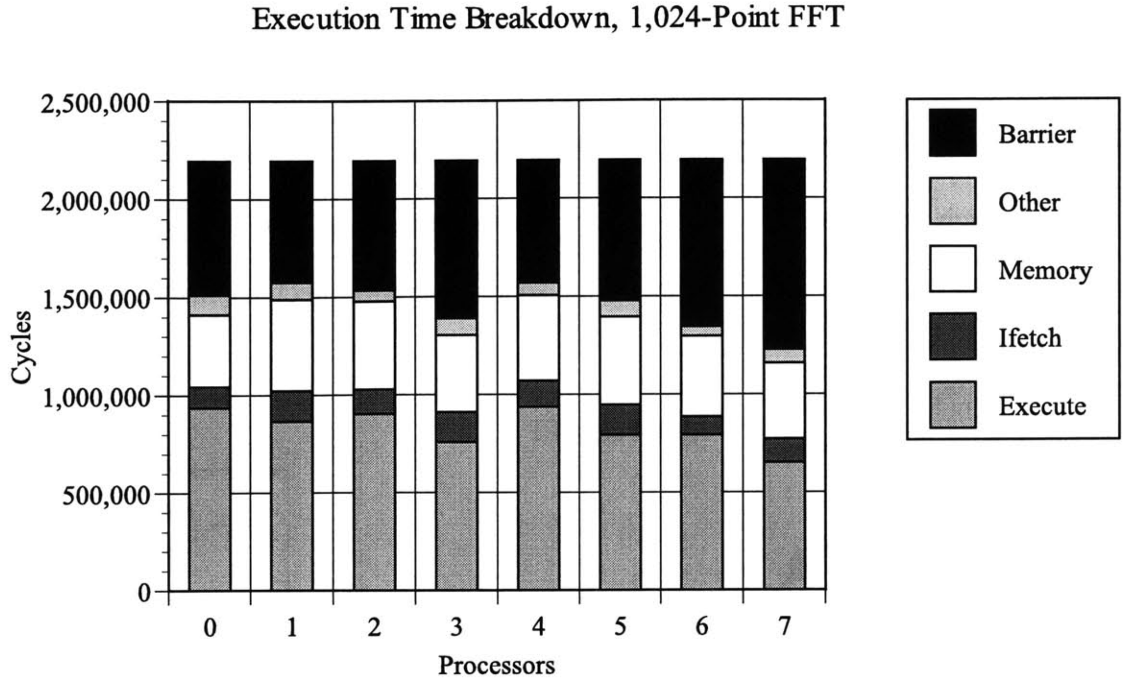


Figure 5.10: Execution Time Breakdown, 1,024-Point FFT

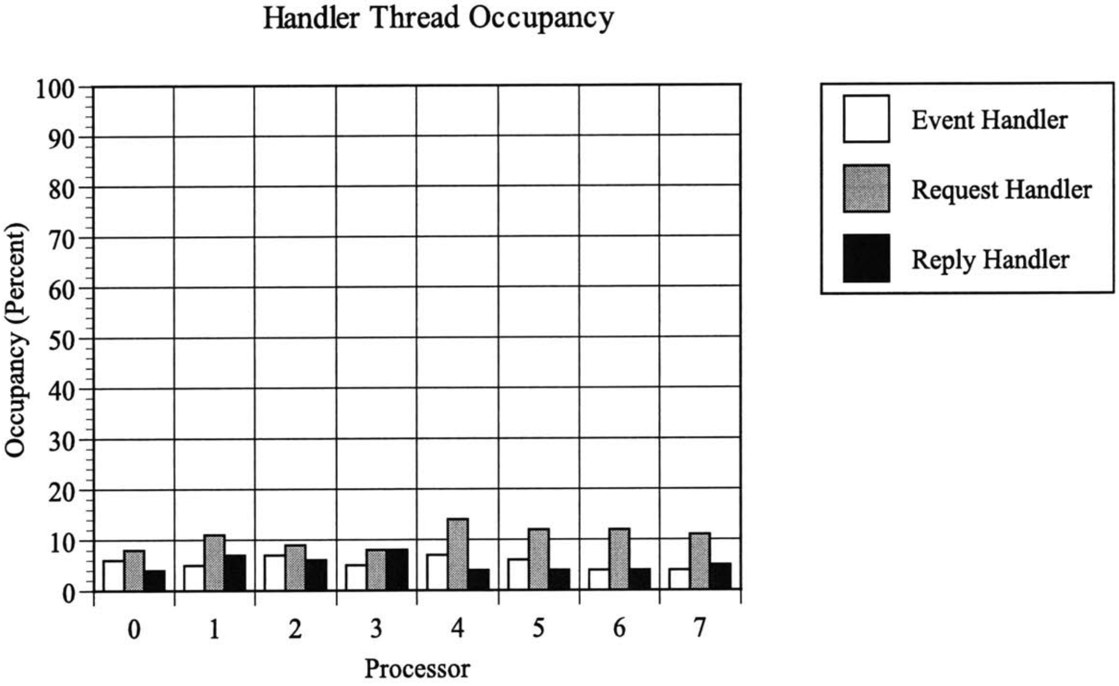


Figure 5.11: Handler Occupancy, 1,024-Point FFT

Figure 5.12 shows how the set of shared-memory mechanisms used affects the execution time of FFT. Surprisingly, the program-level performance of the shared-memory system does not correlate well with microbenchmark-level performance. In particular, the version of the protocol which uses only the block status bits is up to 23% faster than the version which uses all of the M-Machine's mechanisms, in spite of the increase in remote memory access time when only some of the shared-memory mechanisms are used.

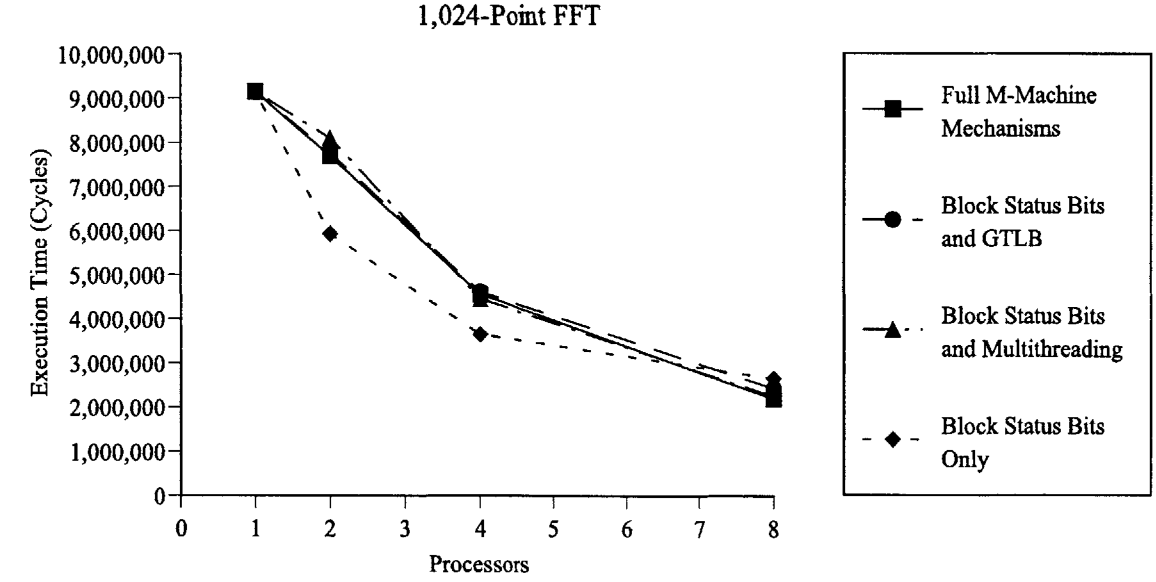


Figure 5.12: Impact of Mechanisms, 1,024-Point FFT

The explanation for this strange behavior is found in the interaction of the shared-memory protocol and miss rates in the LTLB. As shown in Figure 5.13, the amount of time spent handling LTLB misses changes drastically as the set of mechanisms used by the shared-memory protocol changes, overwhelming the effect of remote memory latency on execution time.

Executing shared-memory protocols on the same processor as user threads increases the demands on the address translation hardware by adding an additional stream of mem-

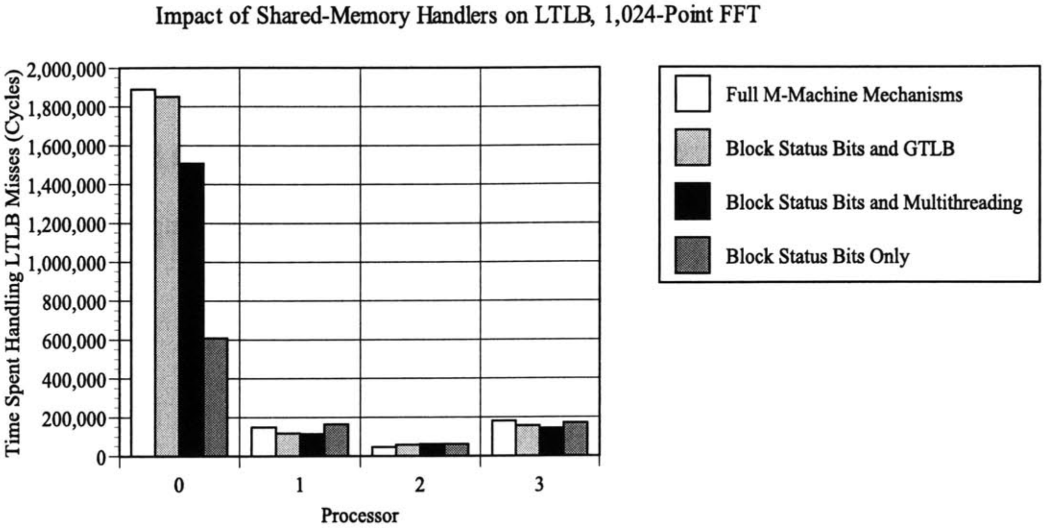


Figure 5.13: Impact of Mechanisms on LTLB, 1,024-Point FFT

ory references which are unconnected to the code and data references made by the user threads. This increases the number of conflicts for TLB entries, and thus the number of TLB misses. On the MAP chip, this effect can be particularly significant because the LTLB is direct-mapped and contains only 64 entries.

Changing the set of mechanisms used by the shared-memory protocol can reduce the number of LTLB misses by increasing the time between virtual-memory references in the protocol handlers. The code and data structures for the shared-memory handlers used in this thesis are physically-addressed, so the shared-memory handlers only reference virtual addresses during block installation and eviction. Because the versions of the protocol which do not make use of all of the MAP chip's hardware mechanisms take longer to resolve remote memory references, they reference virtual addresses less often, since the number of virtual address references per remote memory reference does not change. This reduces the pressure on the LTLB, and therefore the number of LTLB misses.

As shown in Figure 5.14, replacing the MAP chip's 64-entry direct-mapped LTLB with a 128-entry, two-way set-associative LTLB both greatly reduces the number of LTLB misses and the variance in LTLB misses between handlers. The single-node execution time of FFT is reduced by 6%, while parallel execution time is reduced by 13-36%, illustrating the increase in demand on the LTLB which results from executing shared-memory handlers and user code on the same processor.

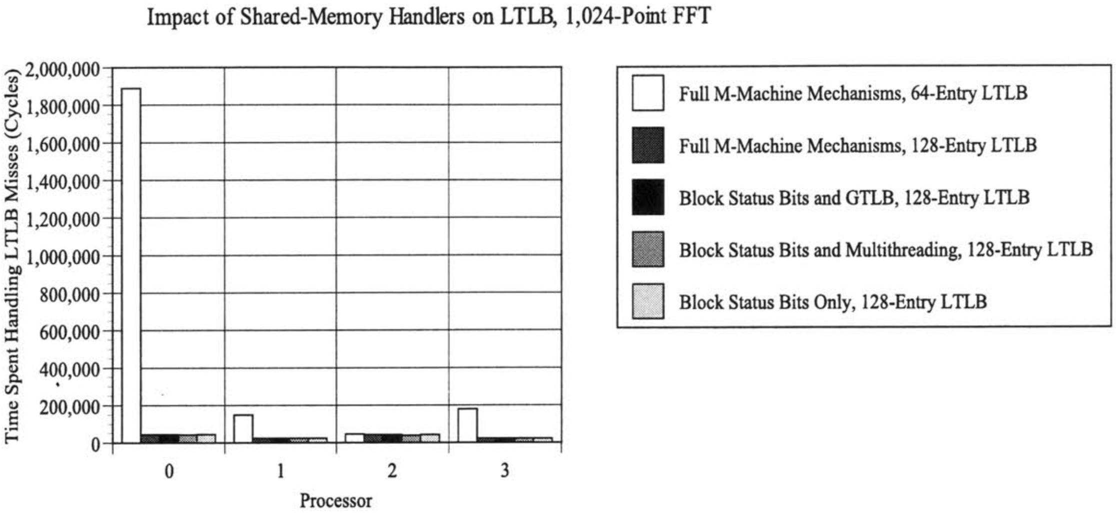


Figure 5.14: Impact of Mechanisms on LTLB, 128-Entry LTLB

When this larger LTLB is used, the program-level impact of the MAP chip's mechansisms is much more consistent with their impact on remote access times, as shown in Figure 5.15. Using only the block status bits and the multithreading scheme reduces performance by 2-4%, while using the block status bits and the GTLB reduces performance by 2-5%. Using just the block status bits gives a 5-9% reduction in performance, virtually identical to the sum of the effects of disabling either the GTLB or the multithreading scheme independently.

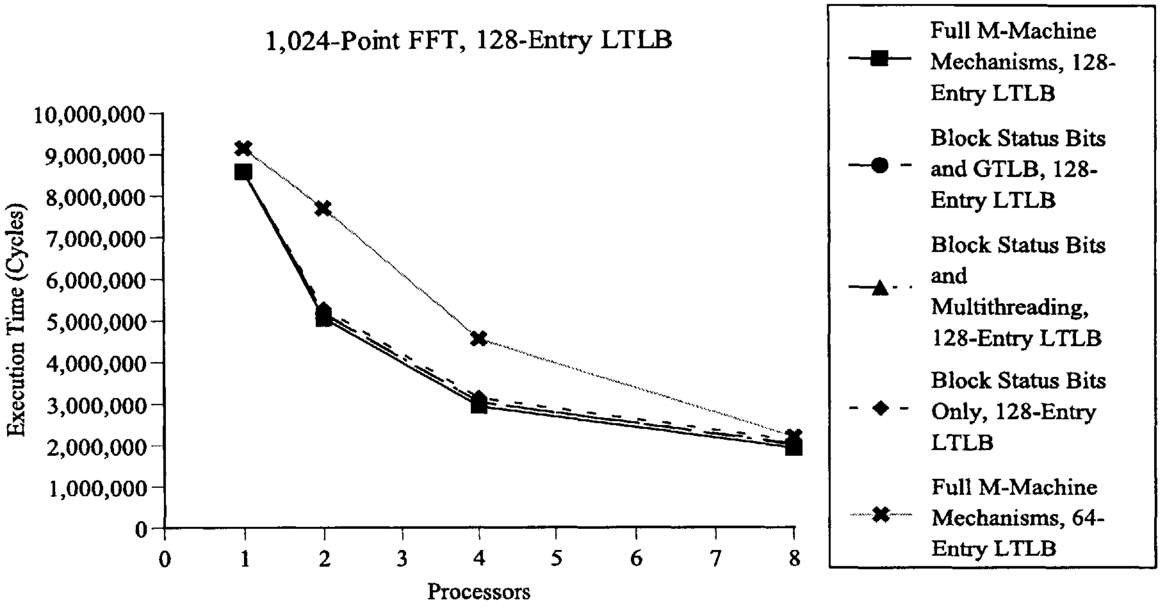


Figure 5.15: Impact of Mechanisms on FFT, 128-Entry LTLB

5.3.2 Multigrid

Figure 5.16 shows the execution time of an 8x8x8 multigrid program on the M-Machine, making use of all the MAP chip's shared-memory mechanisms. Multigrid does not perform nearly as well as FFT on the M-Machine, achieving only a factor of 2 speedup on eight processors. The execution time breakdown shown in Figure 5.17 explains why this program sees so little speedup -- execution time is dominated by memory delays. When running on 8 nodes, the average processor spends only 13% of its time executing instructions, but 54% of its time waiting for memory. Barriers are less of a factor in this program than in FFT, consuming only 22% of the execution time on average.

The occupancies of the shared-memory handler thread slots are shown in Figure 5.18, illustrating a number of important differences between multigrid and FFT. Handler occupancies are higher in general in multigrid, indicating that this program places greater demands on the shared-memory system than FFT. More importantly, the load distribution

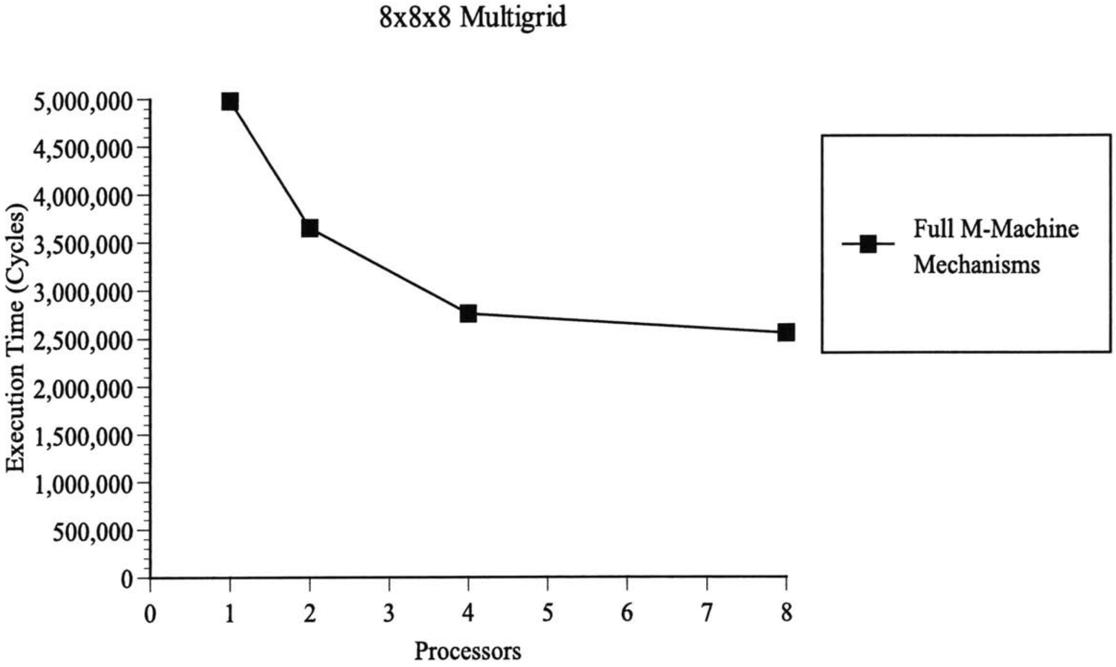


Figure 5.16: 8x8x8 Multigrid Execution Time

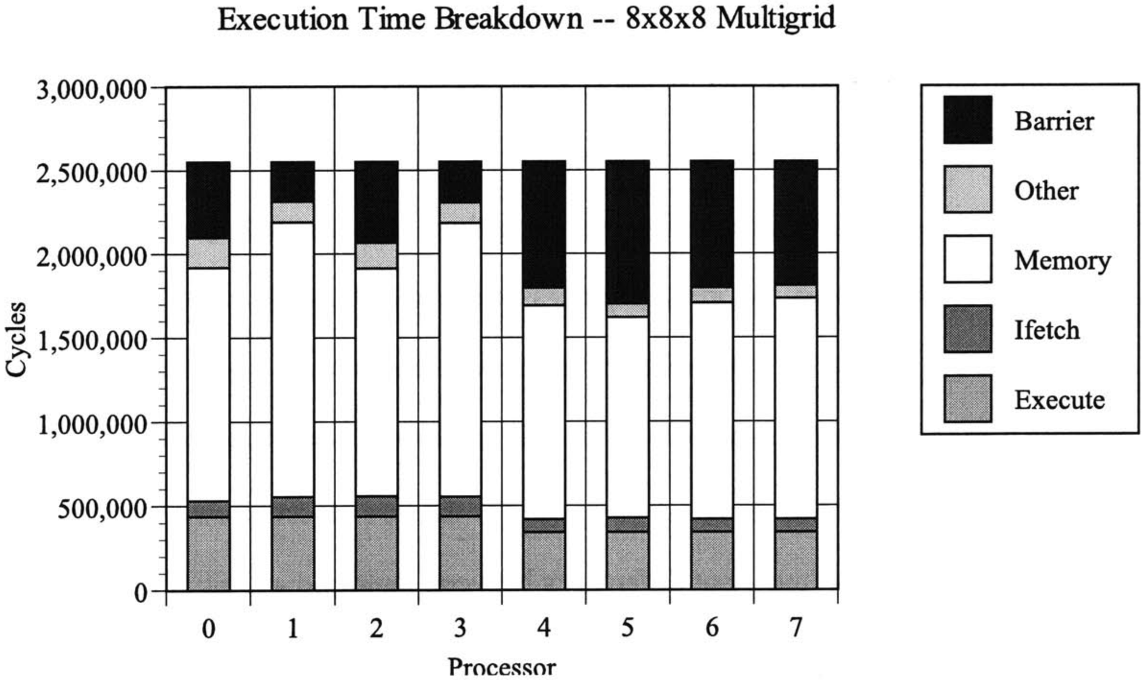


Figure 5.17: Execution Time Breakdown, 8x8x8 Multigrid

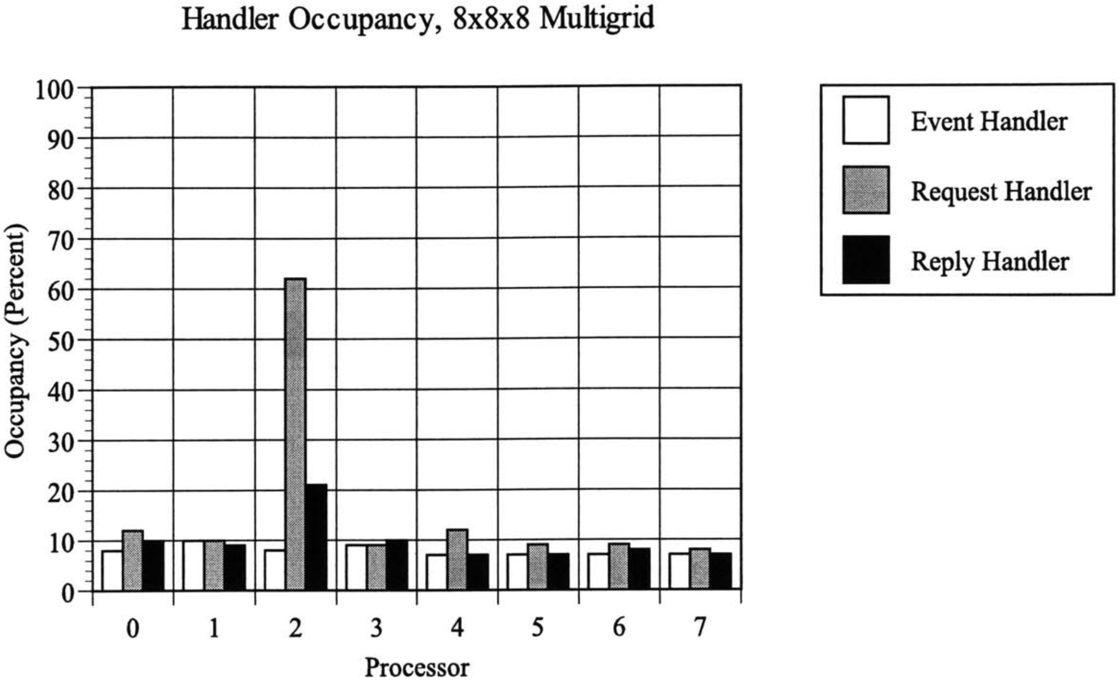


Figure 5.18: Handler Thread Occupancy, 8x8x8 Multigrid

across the nodes is very uneven in multigrid, while FFT showed relatively little variance in handler occupancies. As shown in Figure 5.19, a disproportionate number of remote memory requests target data for which node 2 is the home node, leading to a 62% occupancy rate in the request handler on this node. The high utilization of the reply handler on node 2 is also a consequence of request distribution, as the handler spends a great deal of its time processing acknowledgment messages from data invalidations.

The uneven request distribution is due to the small data structures used by multigrid, and is somewhat an artifact of the small program sizes which it is feasible to run in simulation. An 8x8x8 matrix of 64-bit data occupies 4,096 bytes of storage, equal to the M-Machine's page size and the granularity at which data is allocated to nodes on the M-Machine. This causes the destination matrix of the computation to be mapped onto a single node at this problem size, making that node the hot spot for remote memory refer-

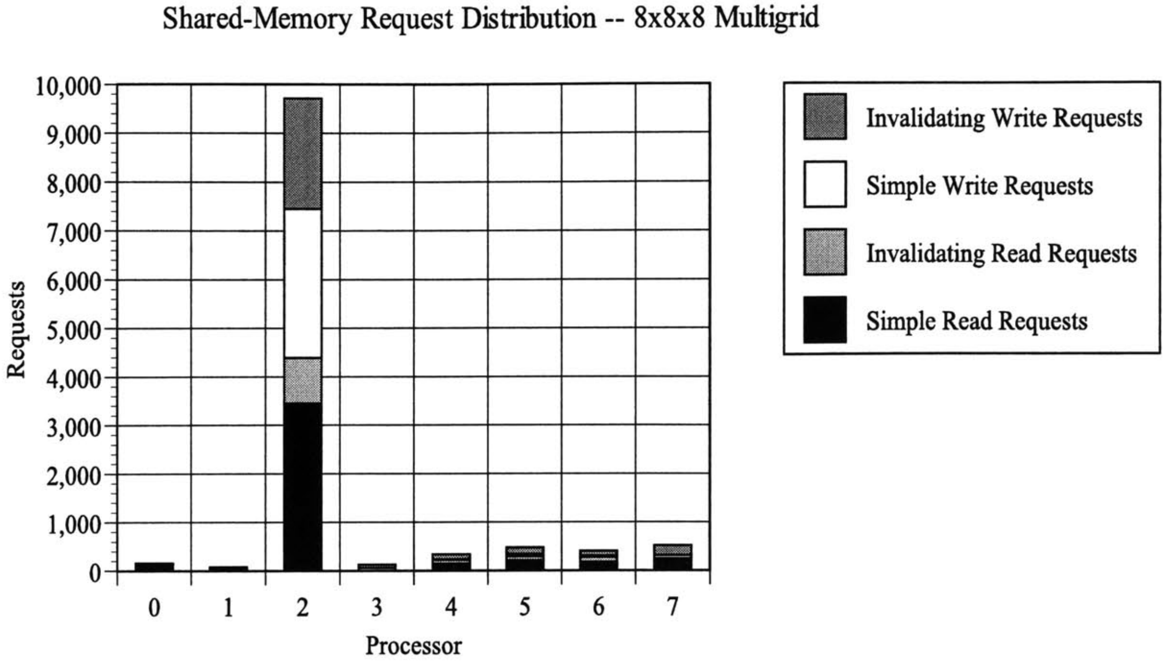


Figure 5.19: Shared-Memory Request Distribution, 8x8x8 Multigrid

ences; and causing the sort of request distribution seen in Figure 5.19. The performance of multigrid could be improved by padding the size of each matrix element so that the matrix occupies several pages of memory, thus reducing the demand on node 2. This was not done in this case, in order to provide an example program which shows how the shared-memory system performs on an application which puts a great deal of stress on the shared-memory handlers.

Based on the results presented above, multigrid would be expected to be more sensitive to changes in the shared-memory protocol than FFT, for two reasons. First, memory delay is a much more significant component of the overall execution time in this benchmark, so changes in the remote memory latency would be expected to have a greater impact than in FFT. Second, the uneven distribution of memory requests makes handler occupancy a significant issue in multigrid. Even with all of the shared-memory mechanisms in use, the request handler thread on node 2 is occupied 62% of the time in an 8-node multigrid, indicating that there is a significant chance that the handler thread will already be occupied when a request message arrives at the node. Therefore, changes to the shared-memory protocol which increase the execution time of the request handler, such as simulating thread-swapping at the start of each handler, will increase not only the latency of the request being handled, but the latency of all requests which are waiting for the handler to complete so they can use the request handler thread slot.

The results presented in Figure 5.20 mostly agree with this intuition, showing substantial performance differences between the different versions of the shared-memory handlers. However, the impact of each of the mechanisms on the program-level performance of multigrid is significantly distorted by the change in LTLB miss rates which result from using different combinations of the MAP chip's mechanisms for shared memory, similar to the effect seen in FFT. As shown in Figure 5.21, a significant amount of time is spent handling LTLB misses when executing multigrid on an M-Machine which incorporates a 64-entry LTLB, and the amount of time varies significantly when different combinations of the MAP chip's mechanisms are used.

As shown in Figure 5.22, going to the larger, more-associative LTLB greatly reduces both the absolute time spent handling LTLB misses and the variance between protocols in multigrid, leading to the execution-time results shown in Figure 5.23. Using just the block status bits increases execution time by up to 30%, while using the block status bits and the GTLB increases execution time by up to 20%, depending on the number of nodes.

Somewhat surprisingly, using only the block status bits and the multithreading scheme does not significantly reduce the performance of Multigrid. In fact, performing address translation in software gives a 1% improvement in performance in the eight-node case. This is because using the GTLB to perform address translation improves the performance

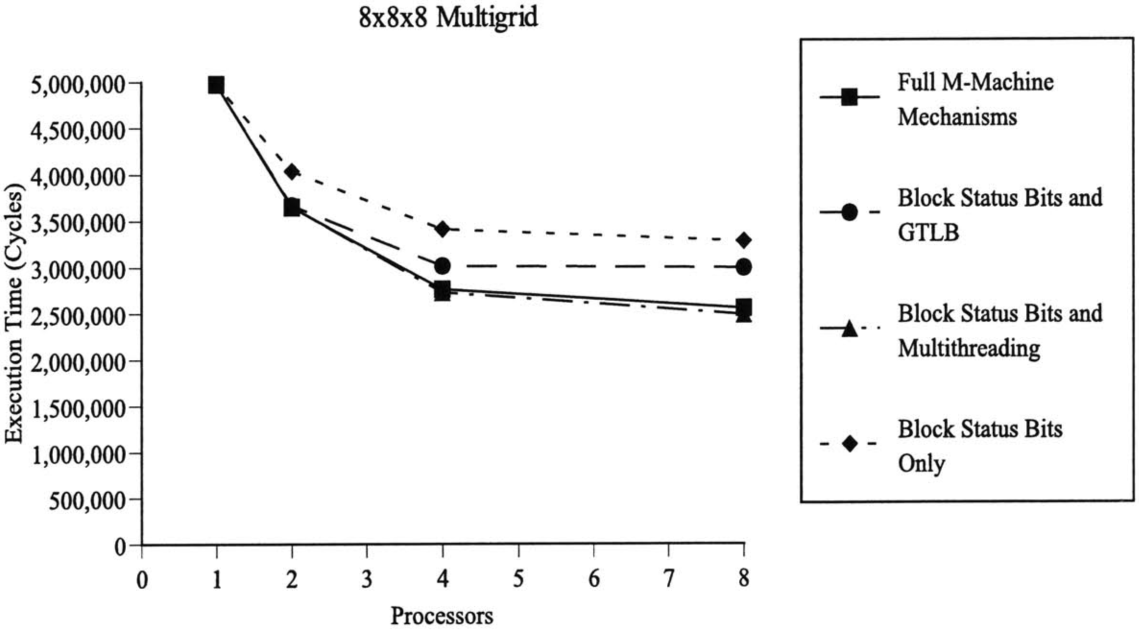


Figure 5.20: Impact of Mechanisms on Execution Time, 8x8x8 Multigrid

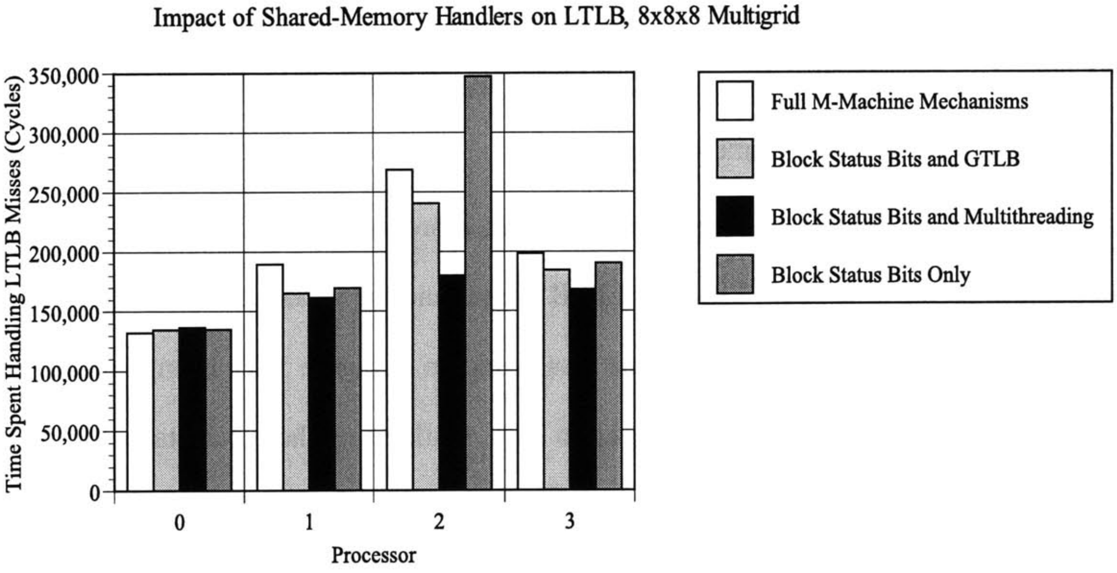


Figure 5.21: Impact of Mechanisms on LTLB, 8x8x8 Multigrid

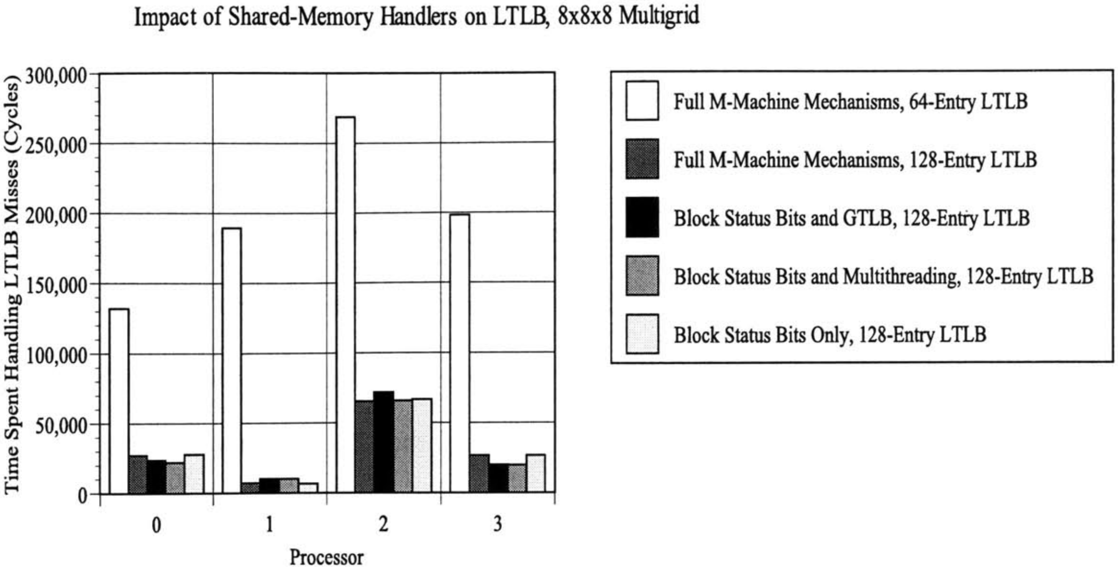


Figure 5.22: Impact of Mechanisms on LTLB, 128-Entry LTLB

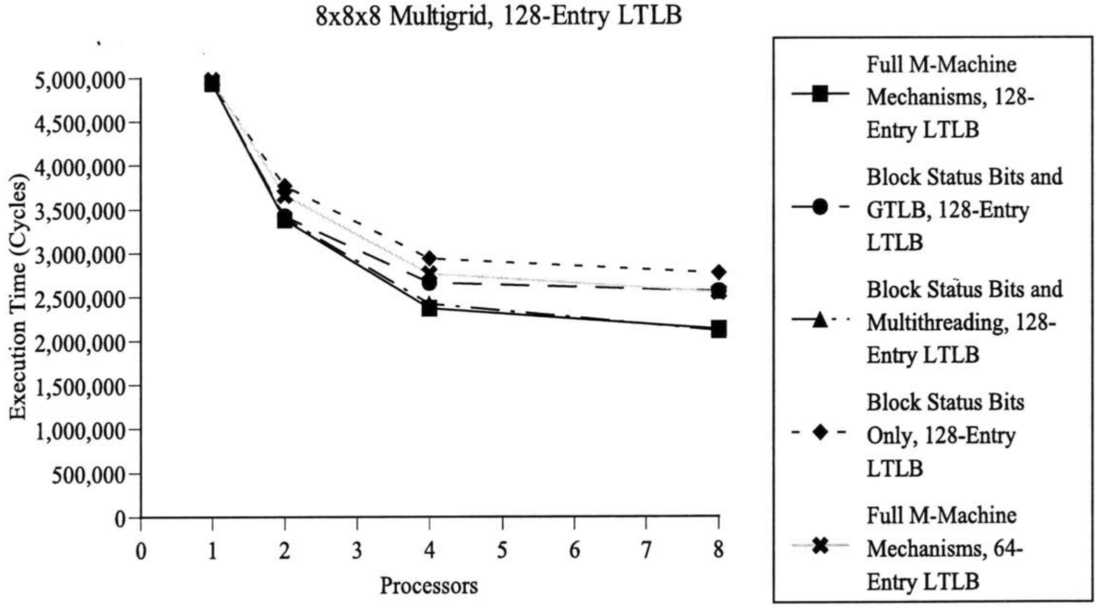


Figure 5.23: Impact of Mechanisms on 8x8x8 Multigrid, 128-Entry LTLB

of the event handler, leaving the request and reply handlers unaffected. On a latency-dominated benchmark like FFT, this improves performance by reducing remote access time. However, multigrid's performance is dominated by the occupancy of the request handler thread on the hot-spot node, which is unaffected by hardware address translation. In fact, improving the performance of the event handler thread can reduce program performance by increasing the rate at which request messages arrive at the hot-spot node, leading to an increase in the number of messages that have to be bounced back to their sender, as shown in Figure 5.24. Since the protocol which uses only the block status bits and the multithreading scheme takes longer to process each event, it sends request messages to the hot-spot node at a lower rate, reducing the number of messages that have to be bounced. Reducing the number of bounced messages reduces the occupancy of the request handler, and thus improves program performance.

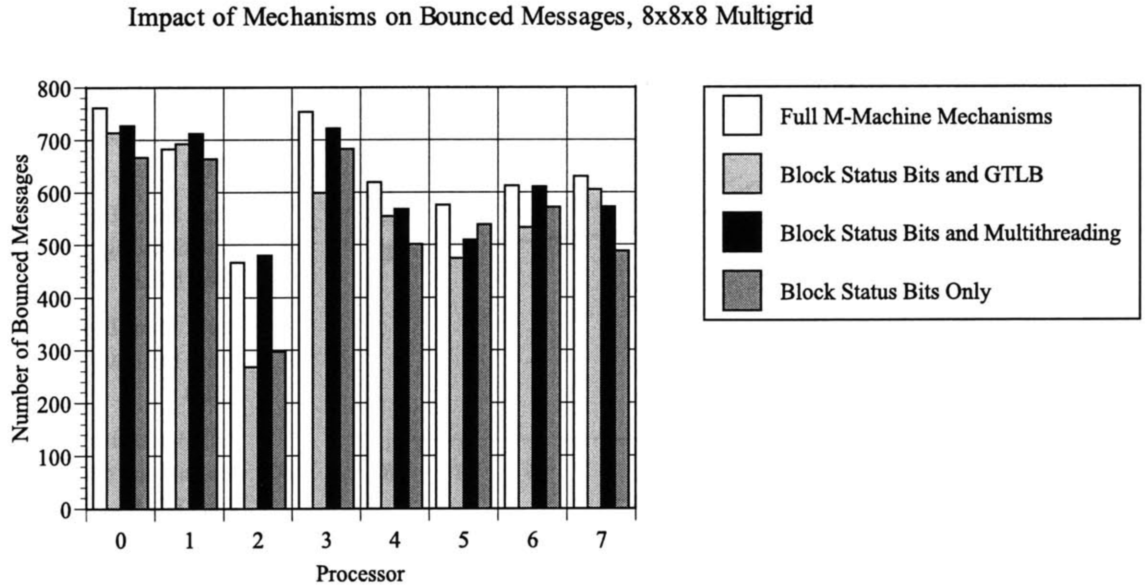


Figure 5.24: Impact of Mechanisms on Bounced Messages, 8x8x8 Multigrid

This chapter has shown that the M-Machine's mechanisms for shared-memory significantly improve the remote access time of software shared memory protocols, at a low hardware cost. The application-level experiments showed that executing shared-memory protocols on the same processing resources as user programs significantly increases the demands on the translation hardware, producing changes in the LTLB miss rate which swamp the reduction in remote access time when run using the MAP chip's 64-entry, direct-mapped LTLB. Using the 128-entry, two-way set-associative LTLB which was originally designed for the MAP, program-level results were much more consistent with the remote access times of the different versions of the shared-memory protocol, showing that the MAP chip's mechanisms improve program-level performance by up to 9% on FFT, and up to 30% on multigrid.

Chapter 6

Additional Mechanisms for Shared Memory

Chapter 5 showed that the MAP chip's mechanisms for shared memory reduce the remote memory access time on the M-Machine by over a factor of four when compared to the M-machine without any of those mechanisms. However, the remote memory latency on the M-Machine is still 2.4x that of a current-generation full-hardware shared-memory system, even with all of the MAP's hardware mechanisms in use. This chapter will describe and evaluate three additional hardware mechanisms which significantly reduce this performance gap: automatic generation of configuration space addresses, instructions which ignore block status misses, and transaction buffers. These mechanisms have not been implemented on the MAP chip, and are presented as candidates for inclusion in future architectures which use similar approaches to shared memory.

6.1 Identifying Areas For Hardware Support

Figure 6.1 shows a breakdown of how time is spent during a remote access when all of the M-Machine's mechanisms for shared memory are in use[1]. From this graph, it is clear that accessing the pending operation structure and evicting and installing blocks are the two largest contributors to the remote access time, and are therefore two prime candidates for additional hardware support. An additional area where hardware support could be effective in reducing the remote access time is the generation of the configuration space address which is used to complete each operation in software. In the base M-Machine architecture, the event handler has to generate this address based on information passed to it by the event system. This process takes about one-third of the time spent in message composi-

[1] The cycles assigned to each category sum to 340 instead of 336, the remote memory latency with all mechanisms in use. This is due to round-off error in assigning cycles to tasks when the handlers pack operations from different tasks into the same instruction.

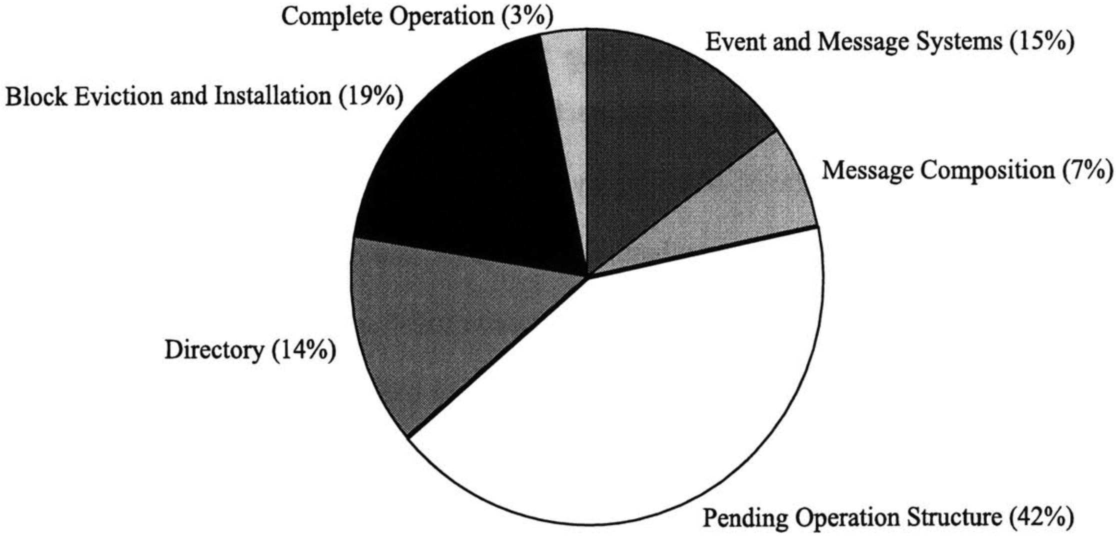


Figure 6.1: Remote Memory Access Time Breakdown

tion, and could easily be eliminated through the addition of control logic to the event system which generates the configuration space address associated with each operation automatically and inserts it into the event queue along with the rest of the event record.

While the time spent accessing the directory and the time spent in the event and message systems are significant contributors to the access time, they are not good candidates for additional hardware support due to the costs involved in providing such support. Considerable effort has been devoted to reducing the latency of the M-Machine's event system, leaving little room for improvement. As is discussed in [21], the latency of the MAP chip's message system could be improved slightly through the use of a streaming network input system instead of requiring that threads compose messages in their register files before sending them, but this would make it much harder for the MAP to guarantee each thread fair access to the network. Providing hardware support for directory accesses is also difficult, since the directory must be large enough to map all of the globally-accessible memory on a node, making it impractical to build an on-chip memory to contain the directory.

6.1.1 Transaction Buffers

The pending operation data structure performs two functions in the cache-coherence protocol: it tracks operations which are waiting for remote data to become available, and it serves as a locking mechanism, preventing multiple handlers from modifying the state of the same block simultaneously. Both of these operations can easily be implemented in hardware, resulting in a significant reduction in remote access time. Transaction buffers, as shown in Figure 6.2, can perform the task of tracking pending operations in hardware. The buffers used for this thesis are fully-associative, storing one operation's information per entry. A set-associative or direct-mapped transaction buffer would be possible, but would require that the hardware be able to handle conflict misses in the transaction buffer. Each entry in the buffer contains the address, data, and opcode of the operation it represents, along with the configuration space address to be used to resolve the operation. In addition, the transaction buffer needs to contain a mechanism for determining the order in which requests to a given block occurred, so that requests can be extracted from the transaction buffer in the order that they entered the buffer to preserve memory ordering for protocols that require it. This can be done either by adding an age field to each entry or by arranging all of the entries that reference a given block into a linked list and having the hardware return the head of the list first.

Figure 6.3 illustrates how the transaction buffer is used by the shared-memory protocol. When the event system detects a remote reference, it creates a transaction buffer entry describing the remote reference, and probes the transaction buffer to see if the buffer already contains an operation which references the same block. If the buffer already contains an operation for the referenced block, the transaction buffer signals the event hard-

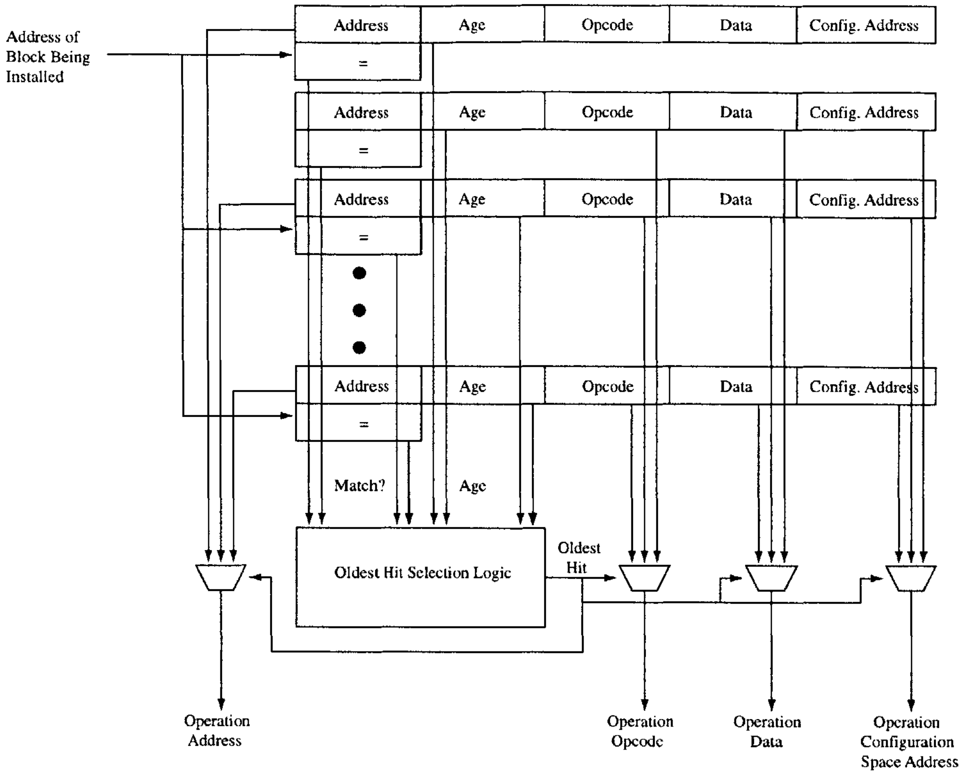


Figure 6.2: Transaction Buffer Block Diagram

ware not to generate an event for the remote reference, reducing the load on the event handler thread[1]. When the remote block arrives at the requesting node, the reply handler executes a new **TPRB** instruction, which either returns the description of the oldest operation which references the block specified by the **TPRB** instruction, or a value which indicates that there are no such operations in the buffer. The reply handler can then use this information to complete the request as if it had read the information out of the pending operation data structure.

[1] In an actual implementation of transaction buffers, this function would be programmable, to allow the implementation of shared-memory protocols, such as uncached protocols, which require that software be invoked on each remote memory access instead of just the first access to each block.

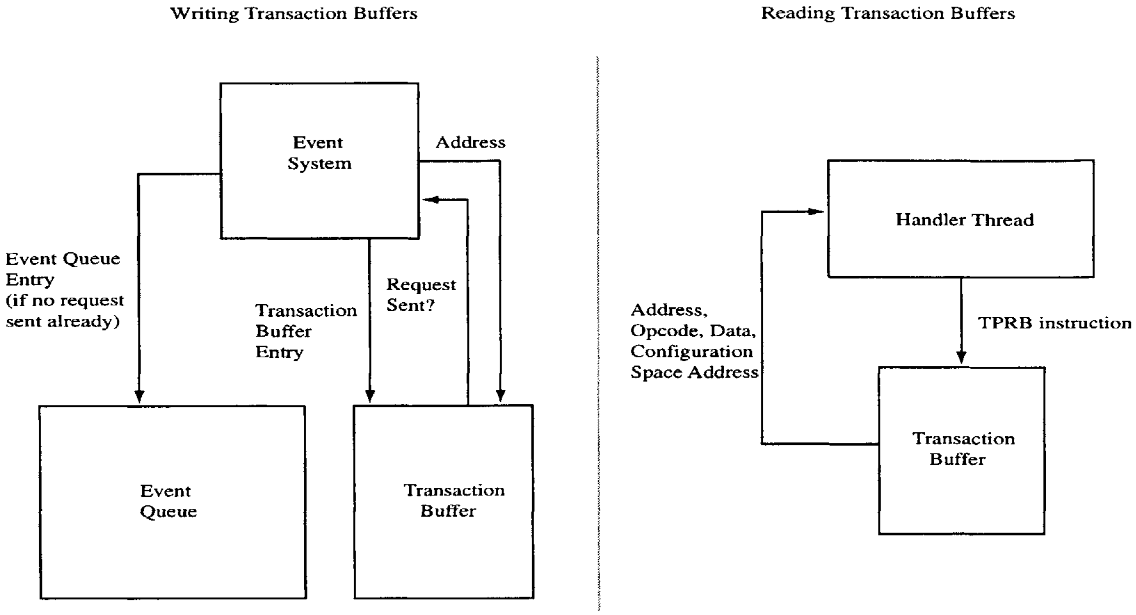

Writing Transaction Buffers
Reading Transaction Buffers
Event System
Address
Event Queue Entry (if no request sent already)
Transaction Buffer Entry
Request Sent?
Event Queue
Transaction Buffer
Handler Thread
Address, Opcode, Data, Configuration Space Address
TPRB instruction
Transaction Buffer

Figure 6.3: Use of the Transaction Buffer

Because the number of outstanding remote data references is limited by the number of operations that the user threads on each node can execute before needing the results of the operations and by the number of transaction buffer entries which the hardware can generate in the time that it takes to perform a remote memory access, the transaction buffer does not need to contain a large number of entries to be able to cache all of the pending remote memory operations. For this thesis, a default transaction buffer size of 32 entries was used, which was large enough that neither of the benchmark programs overflowed the transaction buffer, although the number of transaction buffer entries required would increase if multiple user threads were run on each node or if the compiler were better able to take advantage of the MAP chip's non-blocking memory system to schedule multiple outstanding memory references.

Any transaction buffer implementation requires some mechanism for handling transaction buffer overflows. Architectures which implement an M-Machine style event system can provide this functionality by extending the watermark hardware which halts user threads if the event queue fills up to the point where the memory operations in flight could conceivably use all of the remaining space in the queue to also halt user threads if the number of empty transaction buffers became equal to the number of memory operations in flight. This would increase the number of transaction buffers required since some number of transaction buffers would remain empty at all times, but provides an effective mechanism for handling transaction buffer overflows without increasing the complexity of the shared-memory handlers. If the architecture did not allow this approach, it would be possible to treat the transaction buffer as a cache, using a software pending operation structure to handle transaction buffer overflows, but this would increase the complexity of the software handlers, as they would have to handle the case where some of the pending operations for a block were stored in the transaction buffer and some in the software structure.

6.1.2 Lock Buffers

Lock buffers provide hardware acceleration for the pending operation structure's second function: preventing multiple handlers from modifying the status of a block simultaneously. A lock buffer, as shown in Figure 6.4, contains one entry for each handler which can modify the state of a memory block (three on the MAP), and each entry contains the address of the block of data that its handler is currently modifying. To lock a block, a handler executes a **LOCK** instruction specifying the block to be locked, which probes the contents of all the lock buffers to determine whether any other handler has locked the specified block. If no other handler holds a lock on the block, the lock buffer installs the address of the block in the handler's entry, and returns a value indicating that the operation succeeded. Otherwise, the lock buffer returns a failure value, requiring the handler to execute the **LOCK** instruction again to obtain the lock. An **UNLOCK** instruction clears the valid bit on the appropriate entry in the block buffer, clearing the lock. Since block buffers are allocated to particular system thread slots by the hardware, there is no danger of malicious user programs modifying the state of the lock buffer to interfere with the shared-memory protocol.

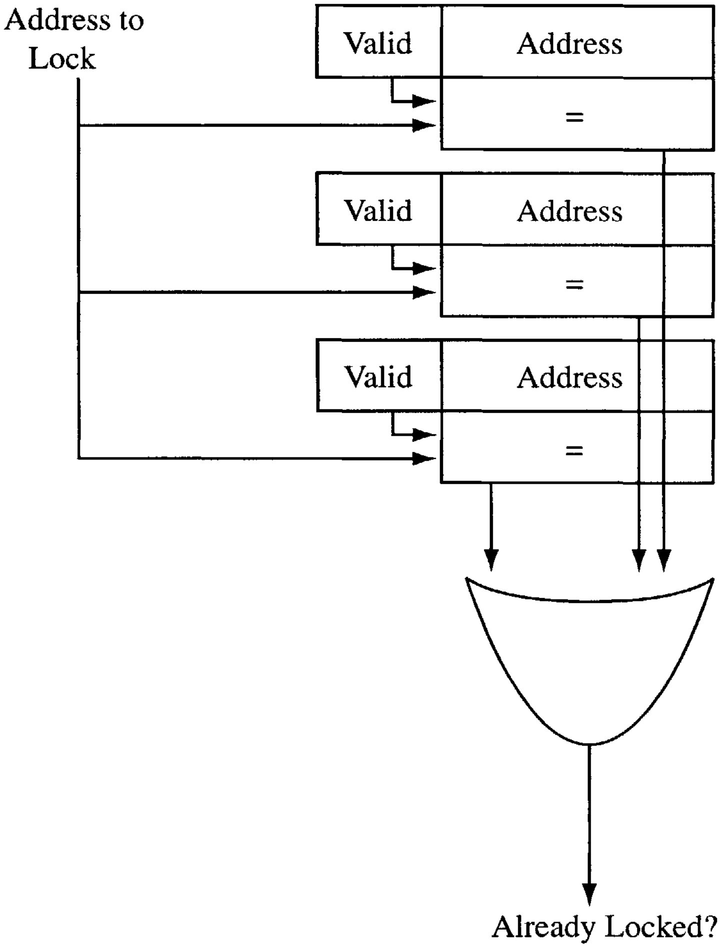


Figure 6.4: Lock Buffer

6.1.3 Instructions That Ignore Block Status Bits

In order to design hardware to reduce the M-Machine's block installation and eviction time, it is necessary to understand why these tasks take as much time as they do. To evict a block from a node, it is necessary to set the status of the block to invalid using the **PUTCSTAT** instruction before the contents of the block are read out of memory to be sent to the requesting node. This prevents write-after-read errors which could result from other threads modifying the contents of the block after the handler evicting the block has read the copy of the block that will be returned to the requesting node, but creates a problem, in that the hardware will not allow the block to be accessed once its status has been set to invalid. Similarly, it is necessary to copy the contents of a block into memory before setting the status of the block to writable or read-only during block installation, to prevent user threads from accessing the block before the most recent copy has been installed, but the hardware will not allow a block to be written if its status is not set to writable.

To get around these difficulties, the shared-memory handlers use the physical address of each cache block to manipulate the block while its block status bits do not allow virtually-addressed accesses to the block. This allows blocks to be installed and evicted safely, but adds a number of cache misses to the installation and eviction process, due to the need to keep the virtually-addressed and physically-addressed copies of each block consistent, as shown in Figure 6.5.

To install a cache block on a node, the handlers must first compute the physical address of the block from information contained in the pending operation data structure for the page containing the block. Then, the block must be copied into the node's memory using the physical address, which causes a cache miss. This cache miss is performed by the event handler as a prefetch, so its impact on the remote memory access time is minimal. Once the entire contents of the block have been copied into the memory, the physically-addressed copy of the block is flushed out of the cache with a flush line instruction to make the virtually-addressed and physically-addressed copies of the block consistent, and a **PUTCSTAT** instruction is used to set the status of the block to indicate that the node now has a copy of the block. Once all of this has been done, the software handlers can use the virtual address of the block to resolve operations which have been waiting for the

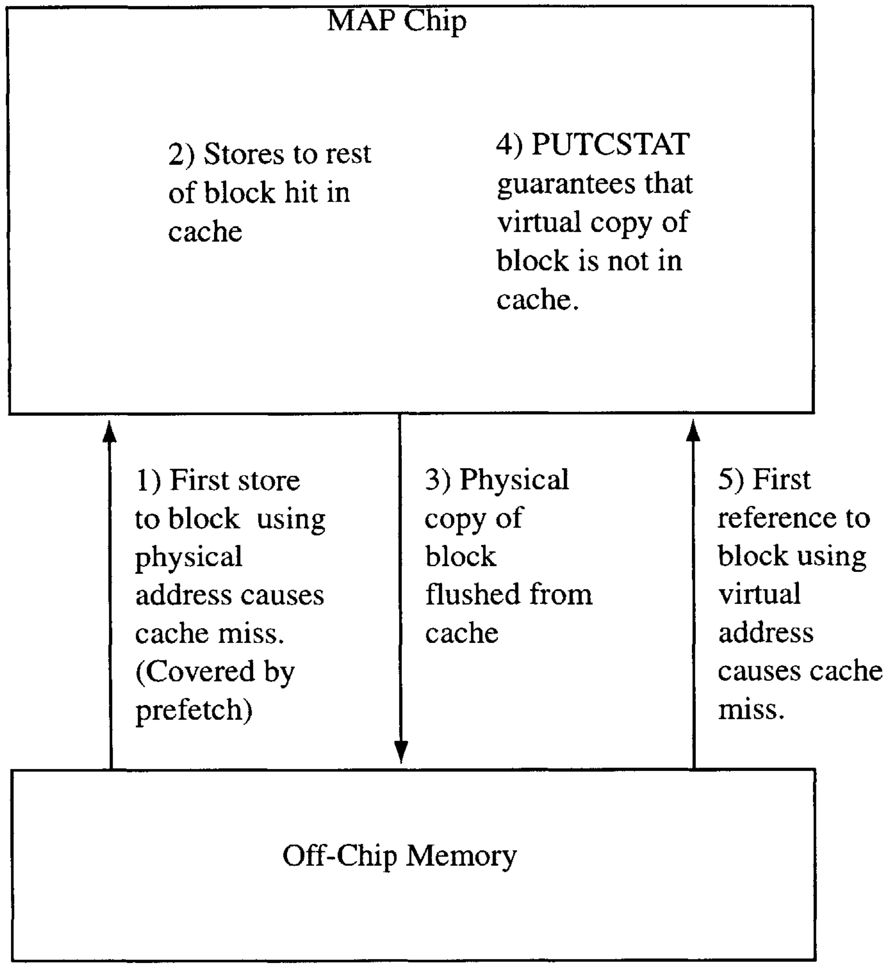

MAP Chip
2) Stores to rest of block hit in cache
4) PUTCSTAT guarantees that virtual copy of block is not in cache.
1) First store to block using physical address causes cache miss. (Covered by prefetch)
3) Physical copy of block flushed from cache
5) First reference to block using virtual address causes cache miss.
Off-Chip Memory

Figure 6.5: Block Installation

block, causing a third off-chip memory access. Evicting a block from a node is basically the reverse process: first the status of the block is set to invalid and the virtually-addressed copy flushed out of the cache, and then the handler uses the physical address of the block to read the copy of the block that will be sent back to the requesting node, requiring two off-chip memory accesses.

If it were not necessary to use the physical address of a block during installation and eviction, it would be possible to eliminate two off-chip memory accesses from the installation process and one from the eviction process (assuming that remote data is rarely in the cache on the home node), since it would no longer be necessary to flush the block out to the off-chip memory to synchronize the virtually-addressed and physically-addressed copies of the block. To make this possible, *instructions that ignore block status bits* could be added to the MAP chip's ISA. These instructions would be privileged, to prevent user programs from interfering with the shared-memory protocol, and would function like normal load and store operations except that they would complete regardless of the block status of the referenced address.

Figure 6.6 shows how block installation would be done if a set of memory instructions which ignore the block status bits were available. The event handler would prefetch the virtually-addressed copy of the block into the cache using the new instructions, similar to the way the base M-Machine event handler prefetches the physically-addressed copy. When the block arrives at the home node, the new instructions could be used to copy the block into memory, and a **PUTCSTAT** operation executed to set the block status bits of the block. The handler could then proceed to use the virtual address of the block to complete the operations which have been waiting for the block, without incurring any cache misses. Similarly, block eviction could be accomplished by setting the status of the block to invalid and then using the new instructions to read the contents of the block into the reply message, incurring either zero or one cache misses depending on whether the block being evicted was in the cache when the eviction started.

In the base M-Machine, **PUTCSTAT** flushes the referenced block from the cache in addition to setting its status to provide an atomic set-and-flush operation to speed up block eviction. If instructions which ignore block status bits were implemented, the behavior of **PUTCSTAT** would have to be modified to not flush the block in order to avoid unneces-

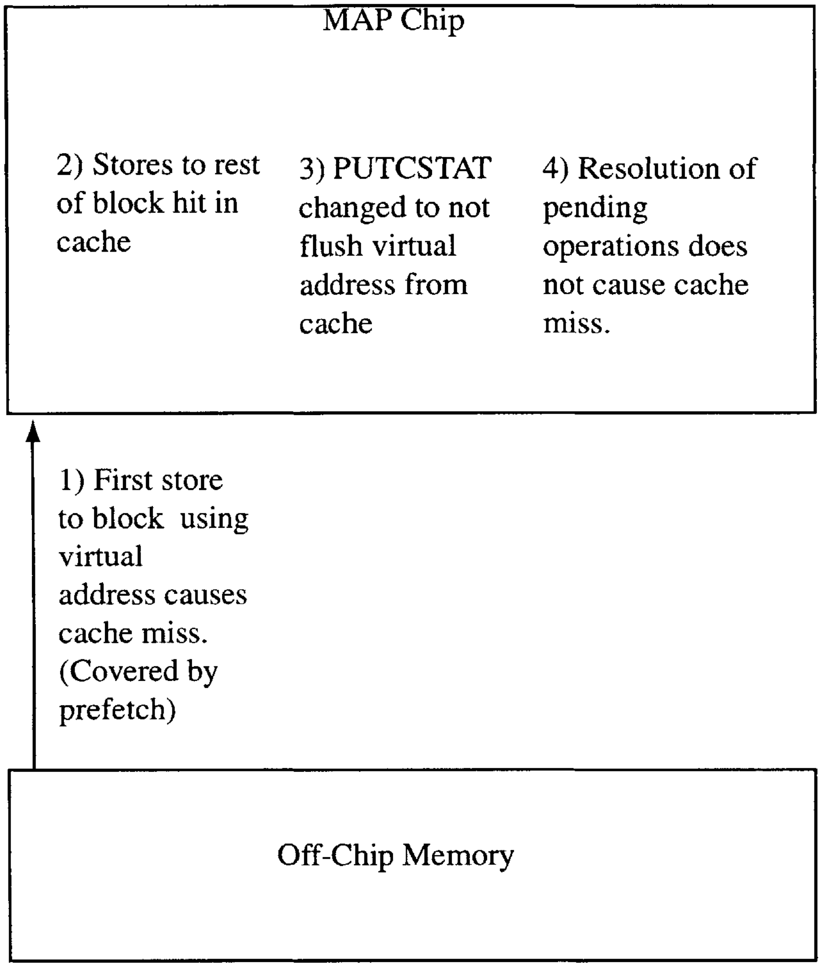

MAP Chip
2) Stores to rest of block hit in cache
3) PUTCSTAT changed to not flush virtual address from cache
4) Resolution of pending operations does not cause cache miss.
1) First store to block using virtual address causes cache miss. (Covered by prefetch)
Off-Chip Memory

Figure 6.6: Block Installation With Instructions That Ignore Block Status

sary off-chip memory references, but the functionality of the current set-and-flush instruction would not be required.

6.2 Implementing the New Mechanisms

Two of the proposed mechanisms, automatic generation of configuration space addresses and instructions which ignore block status bits, require only a small amount of control logic to implement. To add instructions which ignore block status bits to the ISA, the decode logic in each of the clusters would have to be modified to detect the new operations, and the control logic in the memory system modified to allow these operations to complete regardless of the block status of the addresses they reference. Implementing automatic generation of configuration space addresses requires that the event system hardware be modified to generate the bit pattern for the required configuration space address pointer based on the data used to generate the other words of the event queue entry. The main hardware required for this is a single 65-bit register to hold the configuration space address during event queue entry generation.

Adding transaction and lock buffers to the MAP chip requires substantially more hardware. The lock buffer is a conventional three-entry, 49-bit CAM array (48 bits of address[1], plus one valid bit). The transaction buffer is a fully-associative array which requires some custom logic to efficiently update the age field of each entry whenever an entry is added. Each entry in the transaction buffer requires 195 bits of storage (54 for the operation address, 65 for each of the operation data and configuration space address, 6 for the opcode, and approximately 5 for the age/next entry field, depending on the size of the buffer). One relatively efficient way to implement this in hardware is as a 53-bit CAM which holds the 48 address bits which define the block containing the address and the age field, and a 142-bit RAM array. In this configuration, the CAM array is used to select which row of the RAM array gets driven to the output, allowing higher-density SRAM cells to be used for the parts of the transaction buffers which are not required for the address comparison.

The total hardware cost for the proposed new mechanisms is quite reasonable, although greater than that of the base M-Machine's mechanisms. About 6KB of memory is required to implement a 32-entry transaction buffer and 3-entry lock buffer, along with some control logic for the other two mechanisms. This requires about two times the area

[1] The M-Machine uses 54-bit virtual addresses, six bits of which identify the offset within a block, so 48 bits are required to identify a block of data.

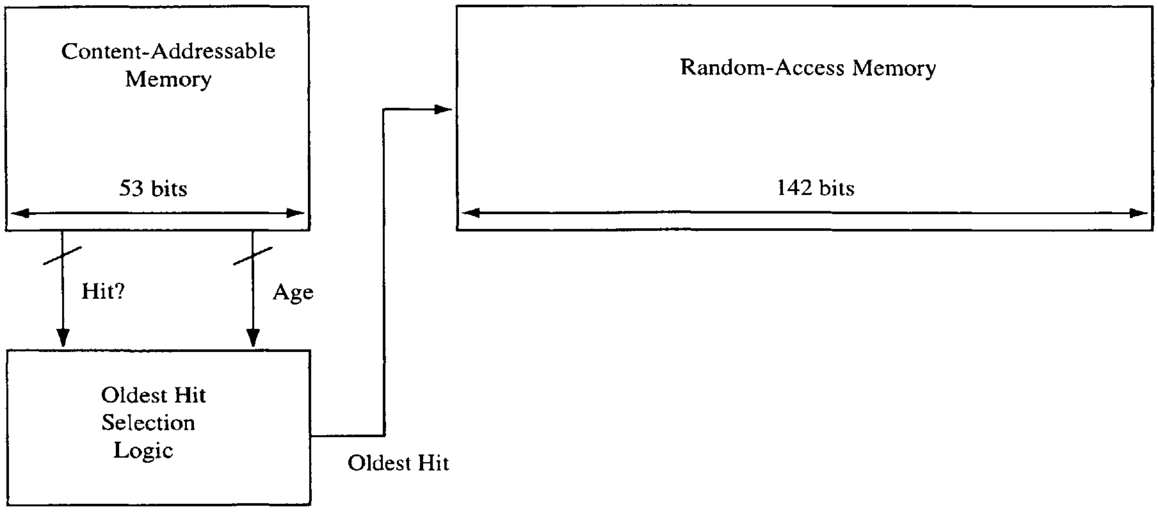


Figure 6.7: Transaction Buffer Implementation

of the MAP chip's current mechanisms for shared memory, and about 20% of the area allocated to its 32KB main cache, although it would only be 5% of the area taken up by the 128KB main cache that was originally designed.

6.3 Evaluating the New Mechanisms

To evaluate the effectiveness of the new mechanisms for shared memory, the MSIM simulator was modified to support these mechanisms and to allow command-line selection of the improvements to be simulated. Three rarely-used instructions were modified to implement the instructions which ignore block status bits and to provide access to the transaction and lock buffers, in order to avoid having to modify the assembler to support new instructions. Additional versions of the shared-memory handlers were then written which take advantage of the new mechanisms, and the same microbenchmarks and application programs which were used to evaluate the M-Machine's mechanisms were run using the modified handlers.

Four different combinations of the new mechanisms were evaluated. Automatic generation of configuration space addresses and instructions that ignore block status bits were evaluated separately and in combination. The transaction and lock buffers were evaluated in combination, since they are intended to address the same issue -- the amount of time spent accessing the pending operation data structure. Also, the evaluation of the transaction and lock buffers assumes that both of the other mechanisms are present. Given the costs of implementing the mechanisms presented in this chapter, any architecture which implements transaction and lock buffers is likely to implement automatic generation of configuration space addresses and instructions which ignore block status bits, which have much lower costs than the transaction and lock buffers.

6.3.1 Remote Access Times

Figure 6.8 shows the remote memory access time of the M-Machine as a function of which new mechanisms the handlers use. From this, we can see that automatic configuration space address generation and instructions which ignore block status bits provide relatively small improvements of 1-5%. Combining automatic generation of configuration space addresses and instructions that ignore block status bits gives a 7% improvement. The transaction and lock buffers are a much more significant improvement, reducing the remote access time by over 32% when compared with the unmodified M-Machine, and 27% when compared with an M-Machine which implements the other two improvements.

Figure 6.9 shows a timeline of a remote reference on an M-Machine which incorporates automatic generation of configuration space addresses. Comparison of this diagram to figure Figure 5.2 shows that adding this new mechanism to the M-Machine reduces the delay from the execution of the operation which causes the remote reference to the execution of the GPRB instruction by 9 cycles, allowing the GPRB to issue on cycle 40 instead of cycle 49. From that point on, the protocol is identical to the full-mechanism protocol on

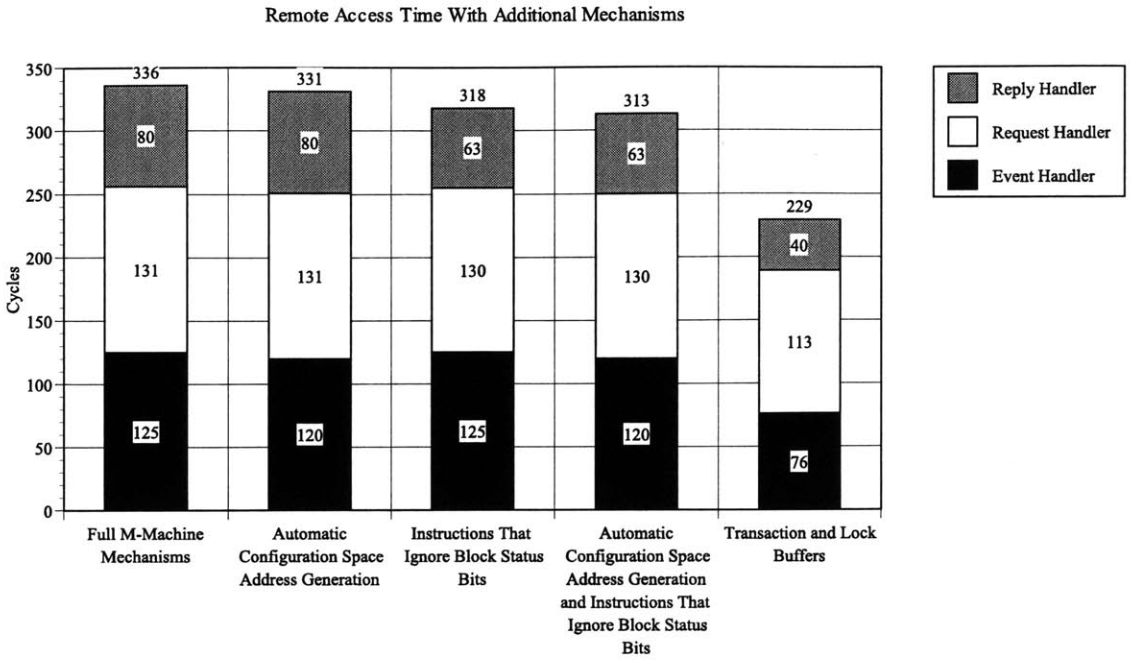


Figure 6.8: Remote Access Times With Improvements

the unmodified MAP, and the handlers proceed almost identically to the original version, completing the remote reference on cycle 331, 5 cycles earlier than the full-mechanism protocol.

Adding instructions that ignore block status bits to the MAP addresses a completely different aspect of the remote-memory delay, as shown in Figure 6.10. In this version of the protocol, the event handler remains unchanged from the full-mechanism version, with all of the changes being in the request and reply handlers. The request handler delay in this example is the same as in the full-mechanism case because the block of data referenced by the operation is not in the cache on the home node, so the request handler still takes a cache miss when reading the virtually-addressed copy of the block into memory, just as the full-mechanism version does. If the requested block were in the cache on the home node, the request handler in this version of the protocol would see a latency decrease, as it

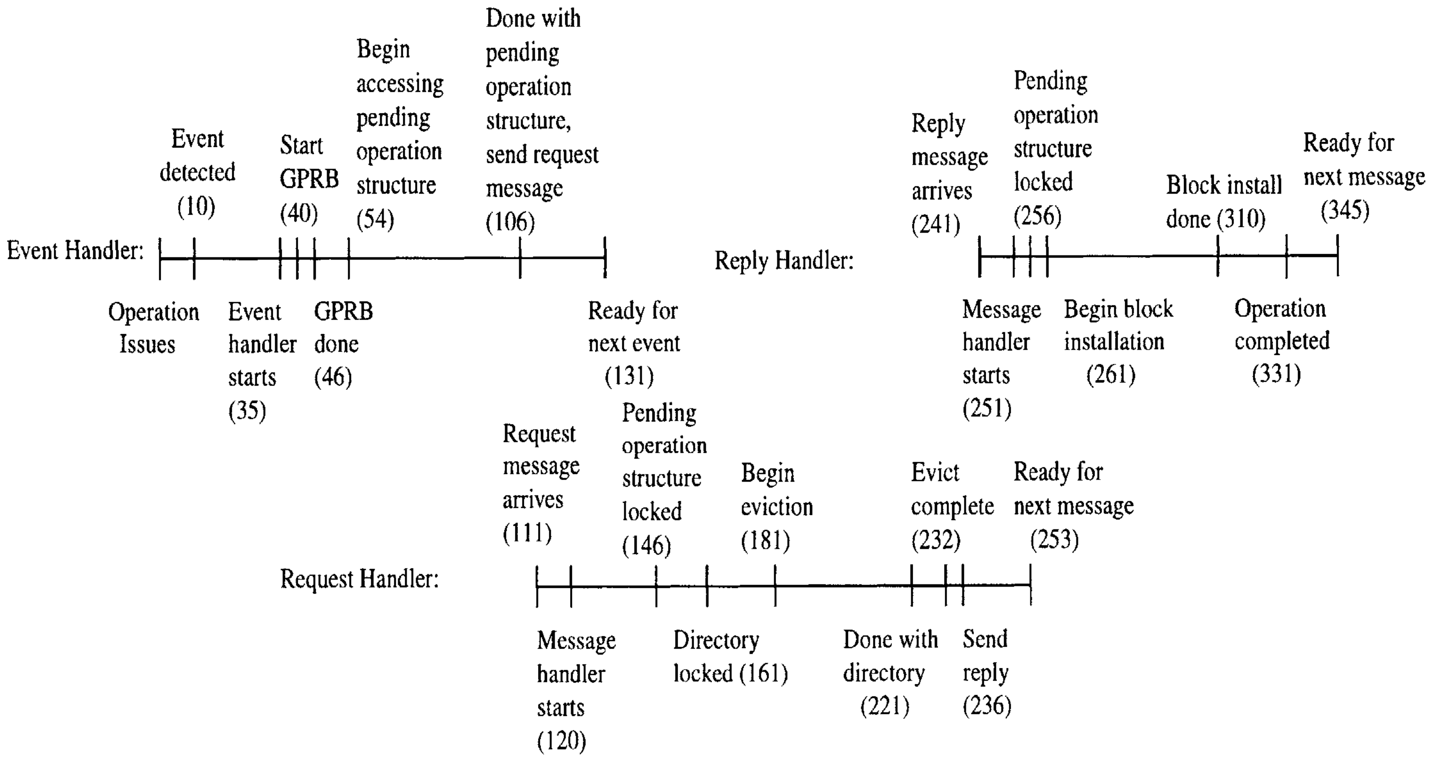

Event Handler:
Event detected (10)
Start GPRB (40)
Begin accessing pending operation structure (54)
Done with pending operation structure, send request message (106)
Operation Issues
Event handler starts (35)
GPRB done (46)
Ready for next event (131)
Reply Handler:
Reply message arrives (241)
Pending operation structure locked (256)
Block install done (310)
Ready for next message (345)
Message handler starts (251)
Begin block installation (261)
Operation completed (331)
Request Handler:
Request message arrives (111)
Pending operation structure locked (146)
Begin eviction (181)
Evict complete (232)
Ready for next message (253)
Message handler starts (120)
Directory locked (161)
Done with directory (221)
Send reply (236)

Figure 6.9: Remote Access With Automatic Generation of Configuration Space Addresses

would not be necessary to fetch the block into the cache, while the full-mechanism version would see a latency increase, as it would have to flush the virtually-addressed copy of the block out of the cache before reading the physically-addressed copy into the cache.

The reply handler delay is substantially reduced by the addition of instructions that ignore block status bits. In the base handler, installation of the requested block begins on cycle 266, and completes on cycle 315, taking a total of 49 cycles. With the new instructions, installation begins on cycle 267 and completes on cycle 302, taking only 35 cycles.

Combining automatic generation of configuration space addresses and instructions which ignore block status bits gives the combination of the benefits of each new mechanism, as shown in Figure 6.11. Hardware configuration space address generation allows the GPRB instruction to be executed on cycle 40, and the instructions that ignore block status bits reduce the block installation time, reducing the total remote access time by 23 cycles.

Adding transaction and lock buffers to the MAP substantially changes the behavior of the shared-memory handlers, as shown in Figure 6.12. Eliminating the need to create a pending operation record for the reference in software allows the event handler to send the request message on cycle 62, 49 cycles ahead of when the full-mechanism protocol sends its request message. Similarly, the request handler sends its reply message 109 cycles after the request arrives, saving 16 cycles because the lock buffer eliminates the need to lock the pending operation structure before proceeding with line eviction. Finally, the reply handler is able to complete the operation which made the remote reference 48 cycles after the arrival of the reply message at the requesting node by eliminating the time required to lock the pending operation structure and retrieve the entry describing the operation so it can be resolved in software.

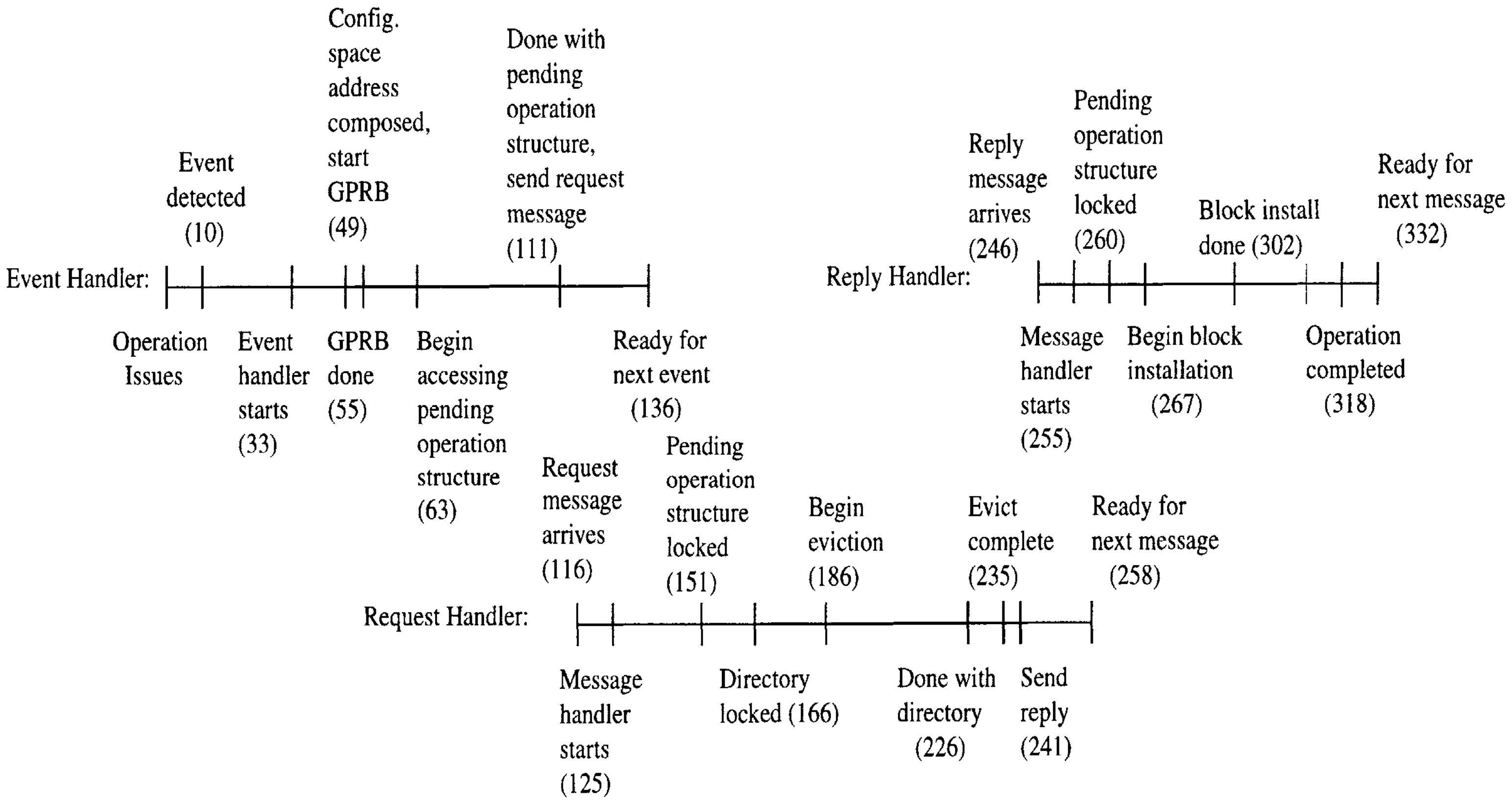

Event Handler:
Event detected (10)
Config. space address composed, start GPRB (49)
Done with pending operation structure, send request message (111)
Operation Issues
Event handler starts (33)
GPRB done (55)
Begin accessing pending operation structure (63)
Ready for next event (136)
Reply Handler:
Reply message arrives (246)
Pending operation structure locked (260)
Block install done (302)
Ready for next message (332)
Message handler starts (255)
Begin block installation (267)
Operation completed (318)
Request Handler:
Request message arrives (116)
Pending operation structure locked (151)
Begin eviction (186)
Evict complete (235)
Ready for next message (258)
Message handler starts (125)
Directory locked (166)
Done with directory (226)
Send reply (241)

Figure 6.10: Remote Access With Instructions that Ignore Block Status Bits

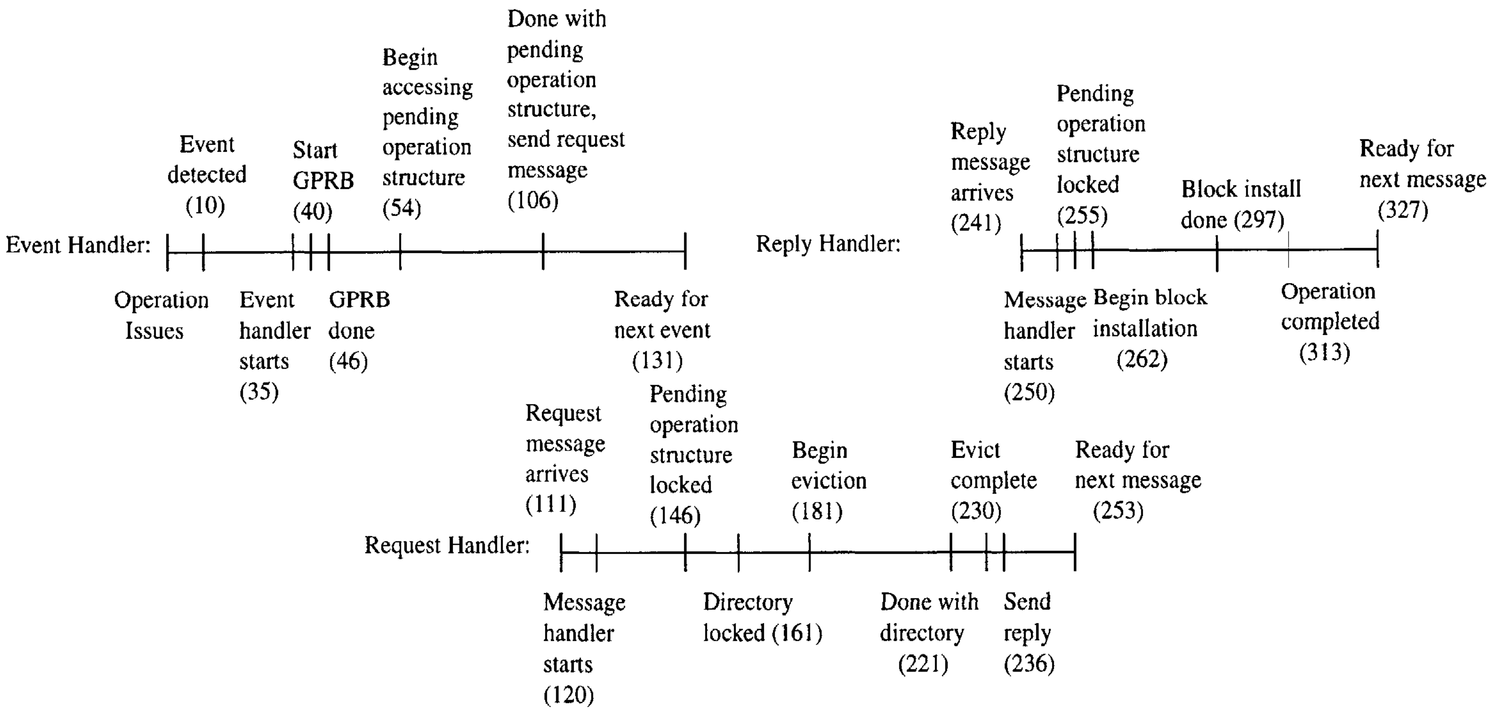

Event Handler:
Event detected (10)
Start GPRB (40)
Begin accessing pending operation structure (54)
Done with pending operation structure, send request message (106)
Operation Issues
Event handler starts (35)
GPRB done (46)
Ready for next event (131)
Reply Handler:
Reply message arrives (241)
Pending operation structure locked (255)
Block install done (297)
Ready for next message (327)
Message handler starts (250)
Begin block installation (262)
Operation completed (313)
Request Handler:
Request message arrives (111)
Pending operation structure locked (146)
Begin eviction (181)
Evict complete (230)
Ready for next message (253)
Message handler starts (120)
Directory locked (161)
Done with directory (221)
Send reply (236)

Figure 6.11: Remote Memory Access with Automatic Generation of Configuration Space Addresses and Instructions that Ignore Block Status Bits

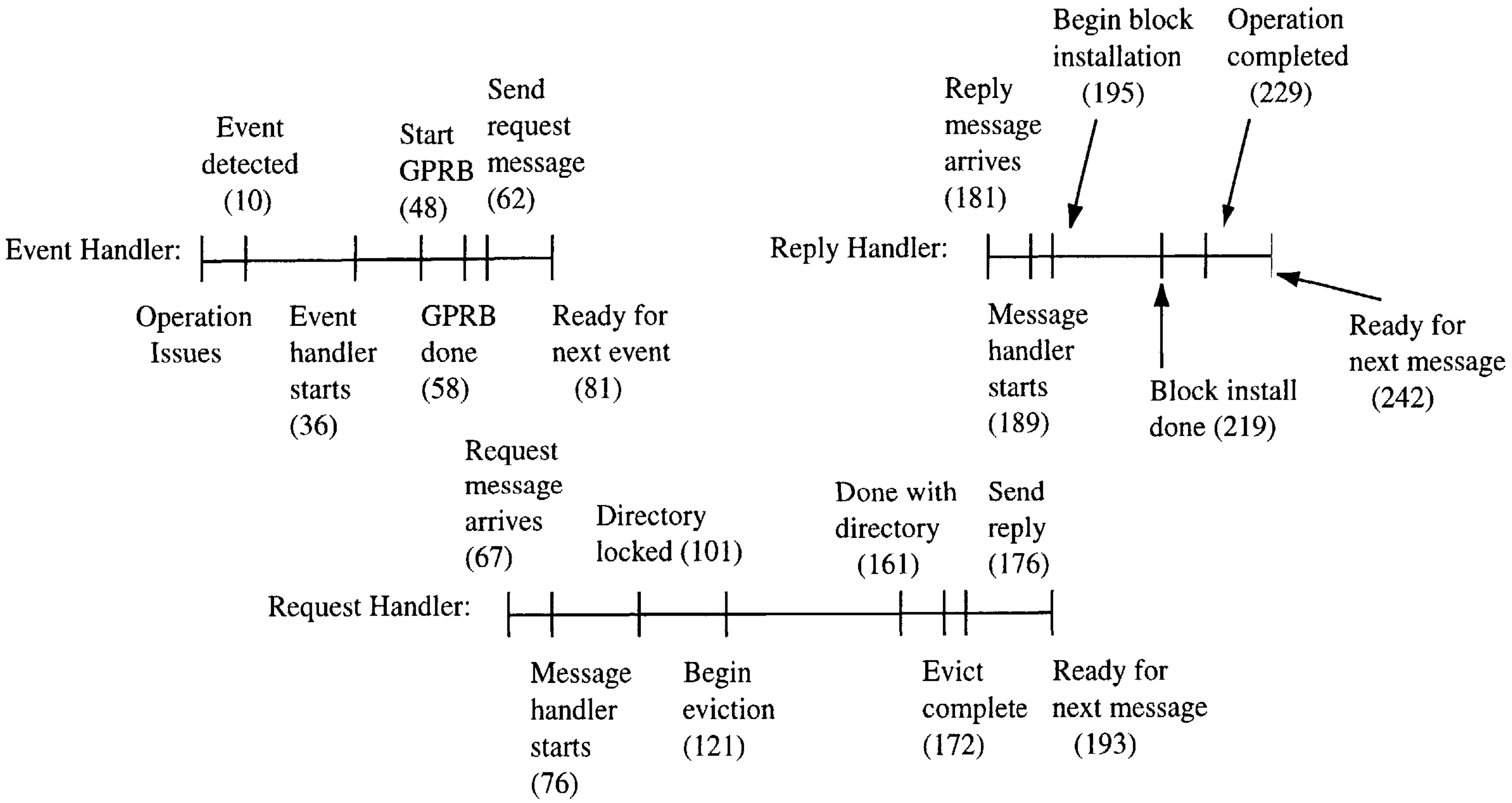

Event detected (10)
Start GPRB (48)
Send request message (62)
Event Handler:
Operation Issues
Event handler starts (36)
GPRB done (58)
Ready for next event (81)
Reply message arrives (181)
Begin block installation (195)
Operation completed (229)
Reply Handler:
Message handler starts (189)
Block install done (219)
Ready for next message (242)
Request message arrives (67)
Directory locked (101)
Done with directory (161)
Send reply (176)
Request Handler:
Message handler starts (76)
Begin eviction (121)
Evict complete (172)
Ready for next message (193)

Figure 6.12: Remote Memory Access With Transaction and Lock Buffers

As shown in Figure 6.13, most of the new mechanisms provide a constant improvement in remote access time, independent of the number of invalidations required to complete the reference. This is because the incremental cost of each invalidation is limited by the time to process data messages from the nodes which have copies of the block being invalidated, which is unaffected by hardware that generates configuration space addresses or instructions that ignore block status bits. The exception to this is the transaction and lock buffers, which reduce the time to process each data message to 37 cycles by eliminating the need to lock the pending operation structure at the start of processing each data message.

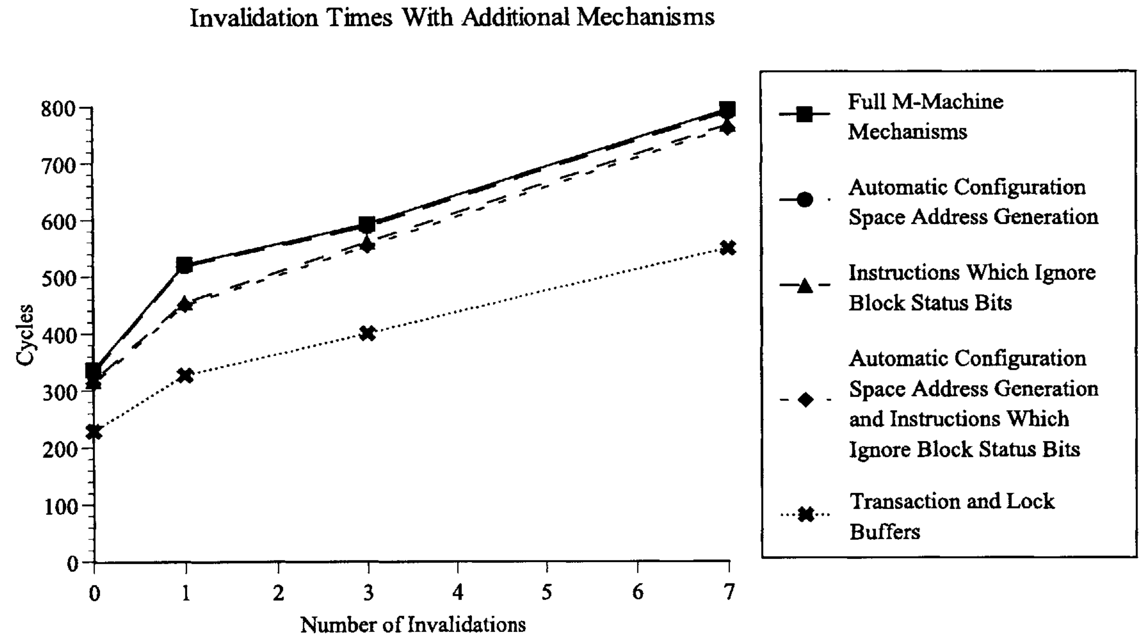


Figure 6.13: Invalidation Time With Improvements

6.3.2 Program Results

Figure 6.14 shows the impact of the new mechanisms on the execution time of a 1,024-point FFT computation. To avoid the effects seen in the last chapter, a 128-entry, two-way set-associative LTLB was used in this simulation. As would be expected from their impact on remote access times, transaction and lock buffers gave the greatest improvement, reducing execution time by up to 9%. Automatic generation of configuration space addresses improved performance by up to 1%, as did instructions that ignore block status bits. Implementing both automatic generation of configuration space addresses and instructions that ignore block status bits improved performance by 1% on two nodes.

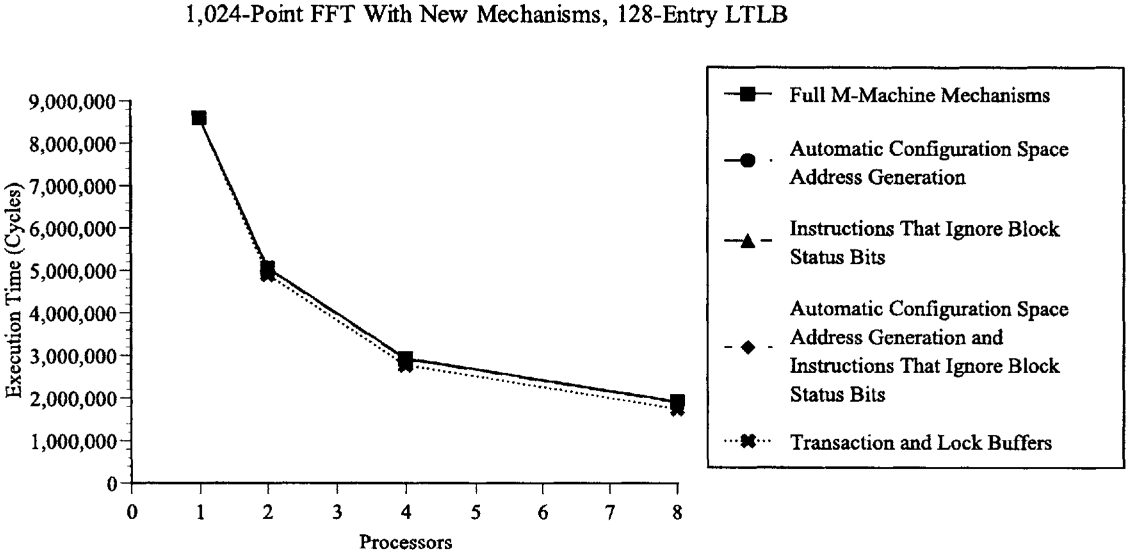


Figure 6.14: 1,024-Point FFT With New Mechanisms, 128-Entry LTLB

Multigrid shows greater sensitivity to the new mechanisms for shared memory, as shown in Figure 6.15. Again, a 128-entry, two-way set-associative LTLB has been used to avoid the LTLB miss effects seen in the last chapter. Transaction and lock buffers provide the greatest speedup, improving performance by up to 16%. Instructions which ignore block status bits give speedups of up to 6%, as does the combination of instructions which ignore block status bits and automatic generation of configuration space addresses.

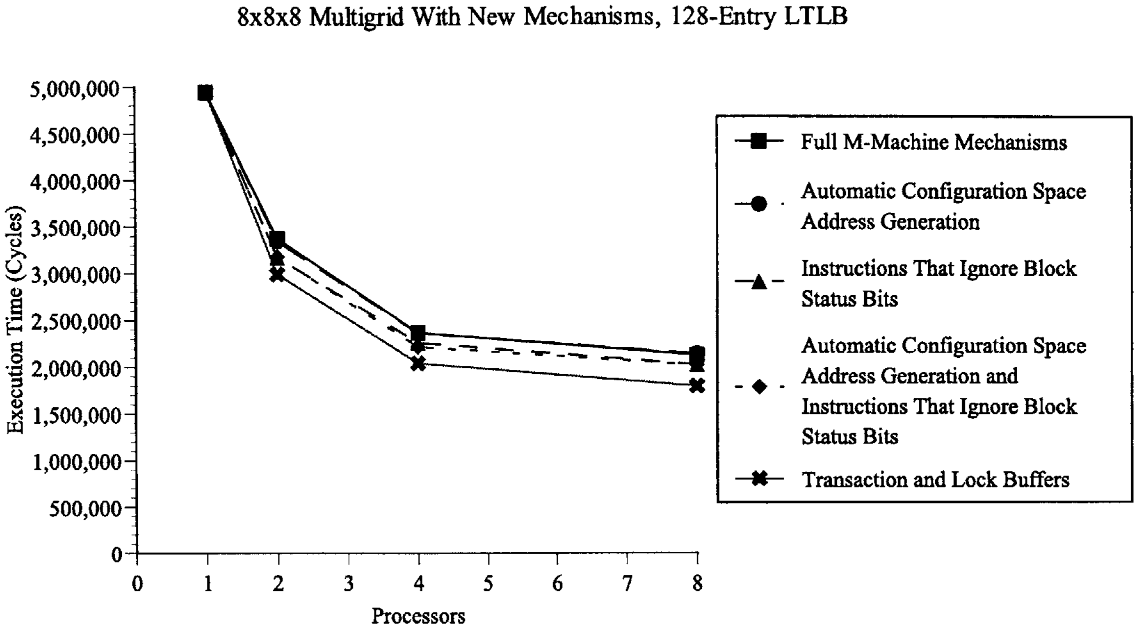


Figure 6.15: 8x8x8 Multigrid With New Mechanisms, 128-Entry LTLB

Automatic generation of configuration space addresses, however, reduces performance by up to 1% on eight nodes, although it improves performance by 1% on two nodes. The explanation for this effect is much the same as the reason why performing address translation in software improved the performance of multigrid, as seen in the last chapter. Generating configuration space addresses in hardware improves remote memory latency by reducing the execution time of the event handler. Since the critical factor for this application is the occupancy of the request handler thread on the hot-spot node, improving the performance of the event handler does not improve the performance of the application significantly. Also, improving the performance of the event handler increases the rate at which request messages arrive at the hot-spot node, increasing the number of messages that have to be bounced back to their sender, as shown in Figure 6.16. This increase in bounced messages increases the load on the request handler on the hot-spot node, reducing overall performance.

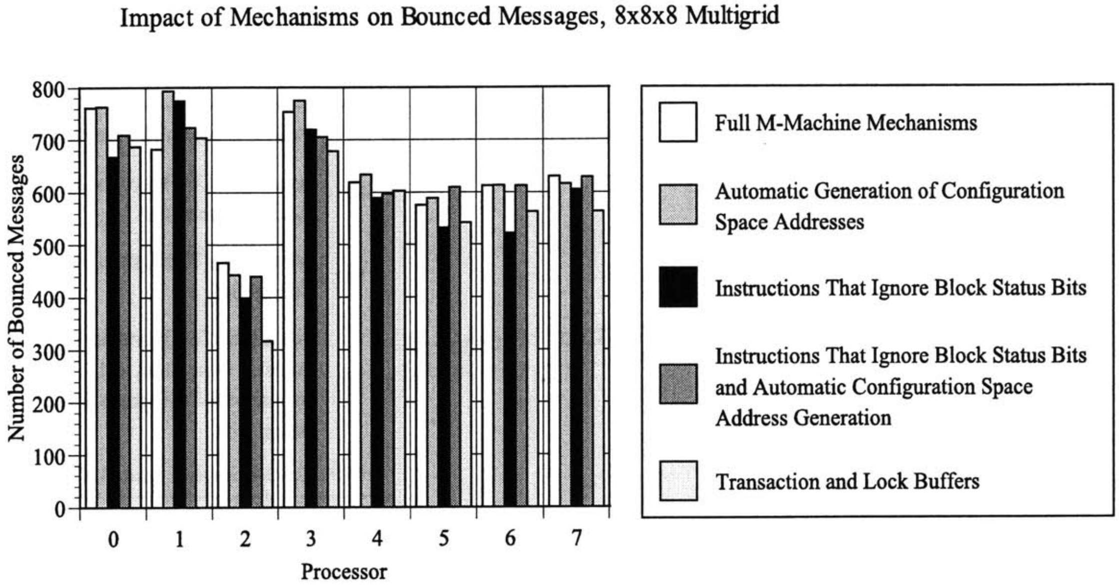


Figure 6.16: Impact of New Mechanisms on Bounced Messages, 8x8x8 Multigrid

6.4 Discussion

The additional mechanisms for shared memory presented in this chapter significantly reduce the performance penalty incurred by using software to implement shared memory. Without these mechanisms, the M-Machine's remote memory access time is 336 cycles, 2.4x that of the SGI Origin 2000. With the additional mechanisms, the remote access time shrinks to 229 cycles, only 64% longer than the Origin's. The new mechanisms have been shown to substantially improve program-level performance as well, reducing the execution time of 1024-point FFT by 9%, and the runtime of an 8x8x8 multigrid program by 16%.

The new mechanisms also address two areas in which the M-Machine's processor architecture reduces shared-memory performance -- the need to use physical addresses to install and evict blocks of data, and the overheads imposed by the non-blocking memory system. The use of physical addresses for block installation and eviction is specific to the MAP processor, but many architectures are exploring the use of non-blocking memory systems to increases latency tolerance. By eliminating the need to use a software data structure to track operations being resolved in software, transaction buffers allow systems to get the latency tolerance benefits of non-blocking memory systems without paying a performance penalty in their shared-memory implementations.

Chapter 7

Comparison to Other Systems

Chapter 2 briefly discussed a number of shared-memory systems in order to provide a context for the discussion of the M-Machine's hardware support for shared memory. This chapter provides a detailed comparison between the M-Machine and several contemporary flexible shared-memory systems to show how the M-Machine's implementation of shared memory compares to other approaches.

7.1 Blizzard and Shasta

Blizzard and Shasta are two recent software-only shared-memory systems which have explored different methods of providing fine-grained shared memory on commodity processors. Blizzard-S and Shasta both detect remote memory references by inserting *check code* into the executable which determines whether or not the data for each memory reference is available in the local memory. This allows small blocks of data to be transferred between nodes, and allows trade-offs between the block size of the shared-memory protocol and the amount of memory required to maintain the state of the blocks on each node. In [32], the Shasta team reports 5-41% increases in sequential execution time as a result of adding these access checks to application programs.

In contrast to these recompilation-based systems, Blizzard-E marks blocks of data invalid by modifying the ECC bits associated with the blocks to force exceptions on any illegal reference to the block, eliminating the overhead of the check code in most cases[1]. Blizzard-S and Blizzard-E both have remote memory latencies of approximately 6000 cycles, while Shasta achieves latencies as low as 4200 cycles, depending on the configura-

[1] Blizzard-E suffers somewhat from spurious exceptions when pages have read-only and writable blocks in them, since the underlying hardware requires that entire pages be marked read-only if any blocks within the page are read-only.

tion of the underlying hardware. Eliminating the software check overhead allows Blizzard-E to achieve significantly better performance than Blizzard-S on most of the applications studied, with the exception of Mp3D, on which Blizzard-S achieved 2% better performance.

Comparing these systems with the M-Machine shows the advantages of the MAP chip's architecture over commodity processors in implementing software shared memory. As was discussed in Chapter 5, the remote memory latency on the M-Machine is estimated to be approximately 1500 cycles when the event system is the only mechanism for shared memory in use, much less than either Shasta or Blizzard. This 2.8x-4x speed advantage comes from a number of factors. The MAP chip's fast network subsystem greatly reduces the overhead for inter-processor communication as compared to processors without integrated network interfaces, while the Guarded Pointers protection scheme reduces context switch overhead by separating protection from translation, and the event system allows invocation of handlers in response to remote references in only 10 cycles.

With the mechanisms for shared memory in use, the M-Machine's advantage in remote access times increases to 12.5x-17.8x over these software-only systems, due to the elimination of both the time required to determine the home node of an address in software and the context-switching overhead involved in invoking software handlers. Adding the new mechanisms proposed in this thesis further widens the gap, allowing the M-Machine to resolve remote memory requests 18-26x faster than Shasta or Blizzard. Clearly, adding a small amount of support hardware for shared memory to a CPU greatly improves the performance of software shared memory at a low hardware cost.

7.2 Alewife

The MIT Alewife machine uses a combination of hardware and software to implement shared memory in order to avoid hardware-imposed limits on the number of nodes in a system. The LimitLESS cache-coherence protocol supports up to five sharing nodes for a block in hardware, and traps to software to resolve cases where more than five processors share copies of a block. Like the MAP, Alewife's SPARCLE processor is multithreaded, using a block multithreading scheme based on the SPARC architecture's register windows to provide multiple program contexts in hardware. SPARCLE also implements a lockup-free cache to allow other threads to continue execution during long-latency memory operations, and the Alewife CMMU chip incorporates transaction buffers to hold the information necessary to resolve pending memory operations when their data becomes available.

The hardware cost for Alewife's shared-memory system is significantly greater than that of the M-Machine's mechanisms for shared memory. In [1], it is reported that the CMMU uses 7363 gates for its network interface, 17399 gates to control the transaction buffer, and 2108 gates for livelock removal, in addition to the SRAM cells required to implement the 16-entry transaction buffer. Also, 2 megabytes of off-chip DRAM memory are dedicated to the directory for the 4MB of shared memory space on each node, an overhead of 50%

When able to resolve remote memory operations in hardware, Alewife has extremely low remote memory latencies, being able to resolve a simple remote read request to an adjacent node in 38 cycles. Requests which require invalidations take 42-84 cycles, depending on the number of invalidations required, giving Alewife almost a factor of 10 advantage in remote memory latency over the M-Machine as a result of its greater investment in shared-memory hardware. If the hardware is unable to resolve a remote request, the memory latency increases to 707 cycles for a request which requires the invalidation of a block shared by six nodes. Read-only requests to widely-shared lines are resolved more quickly than writes, as the shared-memory hardware returns the data to the requesting node before invoking software to update the directory, allowing isolated read requests to widely-shared lines to be resolved at hardware speeds. If a node must process multiple read-only requests to widely-shared lines, it can resolve one request each 425 cycles, as the directory update for a request must complete before the next request can be handled.

Alewife's time to resolve remote memory requests which require invalidation in software is comparable to that of the unmodified M-Machine, which requires 593 cycles to perform a remote access to a block shared by four nodes, and 794 cycles if the block is shared by eight nodes. Since the bottleneck in requests which require multi-node invalidation is the 51 cycles required to handle each data message from a sharing node, resolving a request which requires invalidation of a block shared by six nodes should take 695 cycles on the M-Machine, virtually identical to a six-node invalidate on Alewife. Because the M-Machine uses software to resolve all remote memory requests, read-only requests to widely-shared lines have a latency of 336 cycles, the same as a read-only request to an unshared line. Handling of multiple read-only requests is limited by the execution time of the request handler on the home node, which requires 141 cycles to process a request, update the directory, and be ready for the next request. This gives the M-Machine 3x Alewife's bandwidth for read-only requests to widely-shared nodes.

If the transaction buffers and other improvements suggested in this thesis are added to the MAP chip's architecture, the M-Machine achieves much better software shared memory performance than Alewife, taking 401 cycles to invalidate a block shared by four nodes, 549 cycles if the block is shared by 8 nodes, and an estimated 475 cycles if the block is shared by six nodes (transaction buffers reduce the time to handle a data message from a sharing node from 51 cycles to 37 cycles). Latency for read-only requests to widely-shared lines is reduced to 229 cycles, and the time required to process multiple read-only requests per node decreases to one request every 126 cycles.

Some of the difference between the performance of Alewife and that of the modified M-Machine can be explained by the M-Machine's superior multithreading scheme and use of dedicated thread slots for shared-memory handlers. On Alewife, invoking software handlers requires that the CMMU signal an interrupt to the SPARCLE processor. Servicing the interrupt requires that the SPARCLE processor perform a context switch, which takes 14 cycles for the base context switch, in addition to the time required for the CMMU to transmit the data required by the software shared-memory handler to the SPARCLE. In contrast, the MAP chip is able to respond very quickly to the generation of an event or the arrival of a message in the message queue, without displacing any of the user threads currently executing on the processor.

Alewife achieves very good speedups on shared-memory applications, due to its low remote memory latency when operations are resolved in hardware. On the FFT and Multigrid benchmarks reported in this thesis, Alewife achieves near-linear speedups, showing an 8-processor speedup of 6.6 on the FFT benchmark and 6.9 on the Multigrid benchmark. These speedups are significantly higher than the ones achieved on the M-Machine, although it is difficult to compare the two results because the results reported on the Alewife hardware use much larger problem sizes -- 80K- vs. 1K-point FFT, and 56x56x56 vs. 8x8x8 Multigrid.

Alewife's shared-memory hardware is less flexible than the M-Machine's, having been designed to implement a single shared-memory protocol very efficiently rather than optimized for a variety of protocols. The CMMU allows individual blocks of memory to be marked as "trap always," forcing all requests for this block to be handled in software by the home node, but the process of generating a remote memory request and resolving the reply from the home node is handled completely in hardware. This makes it difficult to implement uncached or update-based shared-memory protocols on Alcwife, as both of these protocols require changes in the manner in which the requesting processor responds to a remote memory reference.

7.3 Typhoon

To provide better performance while retaining the flexibility of software-only shared-memory systems, the Typhoon [29] [30] project at the University of Wisconsin explored the use of a dedicated co-processor to execute shared-memory handlers, eliminating the context switch overhead involved in starting up shared-memory handlers on a software-only system, and allowing handlers to be executed in parallel with user programs. Like the M-Machine, Typhoon allowed remote data to be cached in a node's off-chip memory as well as in the CPU's cache. Three versions of Typhoon were designed: Typhoon-0, Typhoon-1, and Typhoon, which assumed progressively greater amounts of integration in the shared-memory hardware. In Typhoon-0, the user processor, protocol processor, network interface, access control unit, and main memory were implemented as separate modules, connected by a shared bus. In Typhoon-1, the network interface and access controller are merged into a single unit, while the full Typhoon system integrates the network interface, the access controller, and the protocol processor. Typhoon also incorporated a hardware structure known as the RTLB, which contained the access tags for the node's memory. Typhoon-0 and Typhoon-1 did not include hardware to perform this function, relying on software table lookup instead.

Typhoon-0 was implemented in hardware using FPGA technology, and the results used to estimate the latencies for Typhoon-1 and Typhoon. Typhoon-0 required 1461 processor cycles to resolve a remote miss, while Typhoon-1 and Typhoon were estimated to take 807 and 401 cycles, respectively. Comparing these results to the M-Machine, we see that the base M-Machine has noticeably lower remote memory latencies than any of the Typhoon systems, and that the M-Machine's advantage increases to almost a factor of 2 over Typhoon when the new mechanisms presented in Chapter 6 are added. Much of the M-Machine's performance advantage, particularly when compared to the less-integrated versions of Typhoon, comes from its event and network subsystems. Typhoon-0 and Typhoon-1 spend a significant fraction of their remote memory access time detecting remote accesses and invoking software handlers, while the M-Machine and the full Typhoon system are able to start handlers very quickly. In addition, the M-Machine's network has significantly lower latency than the Myrianet used in the Typhoon systems.

7.4 FLASH

The Stanford FLASH machine [19] [13] [14] is another example of an architecture optimized for efficient execution of multiple shared-memory protocols. Like Typhoon, FLASH implements shared memory through the use of a dedicated protocol processor, known as MAGIC. In FLASH, the protocol processor is attached to the memory interface bus of the node's CPU, and handles requests from the CPU for data contained in the node's off-chip memory, data contained in remote memories, and explicit node-to-node communication through messages. One consequence of this design is a that the local off-chip memory latency is increased from 24 to 27 cycles due to the need to relay a request for local data through the MAGIC chip rather than sending it directly from the CPU to the DRAMs. Another consequence is that the MAGIC chip cannot communicate directly with the CPU's cache memory, which is attached to the CPU via a dedicated bus. Because the MAGIC chip must communicate with the CPU to access data in the CPU's cache, the time to handle a remote memory request that hits in the home node's cache is 34 cycles longer than the time to handle a remote memory request which references data in the home node's main memory.

FLASH achieves extremely good remote memory latencies, requiring 111-145 cycles to handle a simple remote read, depending on whether or not the requested data is cached, and 191 cycles to handle a request which requires a single eviction. These latencies are approximately 3x better than the latencies achieved on the base M-Machine, and approximately 2x better than the M-Machine with all of the proposed extensions. FLASH's remote memory latencies also compare quite well with those on the SGI Origin, which requires 140 cycles to perform a remote memory access on an 8-node machine, although the Origin runs at approximately 2x the clock rate of FLASH (5.1ns vs. 10ns cycles). The clock rate difference between the systems is significant, as the protocol processor in the MAGIC chip is only occupied for 11-53 cycles during a remote memory access, suggesting that much of the remote memory latency consists of memory access time and network transmission delay, which are unlikely to improve as the MAGIC processor's clock rate increases. Therefore, it seems reasonable to conclude that the number of cycles required to perform a remote access on FLASH would increase by at least 50% if the clock rate of the MAGIC chip were increased to match that of the Origin, leaving a significant shared-memory performance penalty as a result of adding flexibility to the shared-memory system.

A significant contributor to FLASH's impressive shared-memory performance is the degree to which the MAGIC chip has been optimized to execute shared-memory protocols. Several non-standard instructions have been implemented on MAGIC to improve the performance of shared-memory handlers: find first bit, branch on bit set or clear, a wider set of immediate operations, and insert field operations. In [13], the authors report that at least 38% of protocol processor issue slots contain one of these non-standard instructions, and that most of the instructions are at least a factor of two faster than the code sequences which would be used if the instructions did not exist, suggesting that the latency of shared-memory handlers running on MAGIC is significantly reduced by the addition of these instructions. This intuition was corroborated by an experiment in which the protocol code was re-compiled without using the new instructions and using only one of the MAGIC processor's two issue slots. When programs were re-run using this version of the protocol code, the average program run time increased by 40%, with one program's execution time increasing by 137%, although no breakdown was given showing the individual contributions of the new instructions and the additional issue slot in the MAGIC chip.

The MAP chip implements a number of instructions which are similar to those implemented on MAGIC, although MAGIC's instruction set is much richer. The MAP provides an immediate instruction (**IMM**), which allows generation of a 16-bit immediate, and a shift-and-or (**SHORU**) instruction, which shifts a register by 16 bits and then ORs the result with a 16-bit immediate contained in the instruction. These instructions allow 64-bit immediates to be assembled in four cycles, and are implemented in both the integer and floating-point units. These greatly improve immediate generation on the MAP, but still require several cycles to generate some of the mask fields used in the shared-memory handlers. Immediate generation instructions like the ones implemented in FLASH, which allow generation of bit fields containing all ones or all zeroes in a single instruction, would have been very useful in implementing shared memory on the M-Machine.

The MAP does not implement find-first-one or branch-on-bit instructions as MAGIC does. It does provide a set of bit-field insertion instructions, but these instructions are somewhat limited in that they only allow insertion of 8- or 16-bit quantities into words, which must be inserted starting at a bit position which is a multiple of their size. A more generic shift-and-insert instruction would have been very useful in implementing the shared-memory handlers, particularly in manipulating the pending operation structure and the directory.

One limitation of FLASH is that it does not directly support caching of remote data in the off-chip memory of a node as the M-Machine does, limiting the amount of remote data present on a node to the size of the CPU's cache. This limitation can be overcome in software by having the MAGIC chip probe a directory to see if a requested address has been cached in the off-chip memory, although this will increase the memory latency for all memory requests.

The FLASH project has shown that optimizing a co-processor for execution of shared-memory handlers can significantly improve the performance of a flexible shared memory system. Even greater performance could be obtained through a combination of the M-Machine's mechanisms and the techniques developed for FLASH. Extending the MAP's instruction set to support the instructions developed for FLASH would reduce the execution time of the shared-memory handlers while still allowing off-chip memory requests to be resolved in hardware, and would also eliminate the performance penalty seen on FLASH whenever the MAGIC chip needs to access the CPU's cache.

Chapter 8

Future Work

A number of projects, including Tempest [29], Shasta [32], and Cashemere [36] have demonstrated that providing flexibility in a shared-memory computer system can improve performance. However, all of these systems share two drawbacks. First, they rely on software to implement some or all of their shared-memory protocols, which increases the remote memory latency. Second, they require effort from the programmer to select an appropriate shared-memory protocol for each application.

The mechanisms presented in this thesis reduce the overhead of software shared memory substantially from the 1500+ cycles required for a remote access on an M-Machine without any hardware support for shared memory to 229 on an M-Machine which incorporates all of the proposed extensions as well as the MAP chip's mechanisms for shared memory. However, even with all of these mechanisms in place, the remote memory latency is still more than 50% greater than the 140 cycles required for a remote access on the SGI Origin 2000, which implements shared memory completely in hardware. To achieve the maximum benefit from flexible shared memory, it is necessary to close this performance gap by designing full-hardware flexible shared-memory systems, capable of executing multiple protocols at hardware speeds.

One approach to this would be to implement several protocols in hardware, and allow protocols to be selected on either an application-by-application or data structure-by-data structure basis. However, this would limit the number of protocols that the system could support, and would require that each protocol be tested and proved correct, greatly increasing the design time for the system. A better idea would be to continue the M-Machine's approach of using programmable control of hardware which implements lower-level functions by implementing hardware to perform all of the functions involved in shared-memory protocols, such as probing a software data structure to determine if a request has already been sent for a block, sending request messages, evicting and installing blocks, accessing software directories, and so on. A programmable state machine could then be used to compose various shared-memory protocols out of these functions by sequencing through them.

Figure 8.1 shows how this system could work. An ID associated with each region of memory would be used to access a memory array, producing the state-transition table to be used to implement the shared-memory protocol for that region of memory. The initial state for the finite-state machine would be a function of the action being performed (responding to a reference to remote memory, handling a request message, etc.). Once the state-transition table and the starting state had been determined, the finite-state machine would iterate through the appropriate set of actions until reaching the termination state, at which point it would be free to handle the next request. For greater performance, multiple state machines could be implemented in hardware, allowing remote references, request messages, and reply messages to be processed in parallel.

Using a programmable state machine to implement shared memory essentially provides the best of both worlds, allowing the system to support a wide variety of protocols at hardware speeds, while reducing the verification effort by dividing the shared-memory hardware into independent mechanisms which can be tested separately. However, such a system would still require that the programmer select the shared-memory protocol or protocols to be used for each application. In general, systems which offer improved performance at the cost of increased programmer effort have been of limited success, suggesting that having the either the hardware or the compiler select the shared-memory protocol to be used would greatly increase the attractiveness of a flexible shared-memory system.

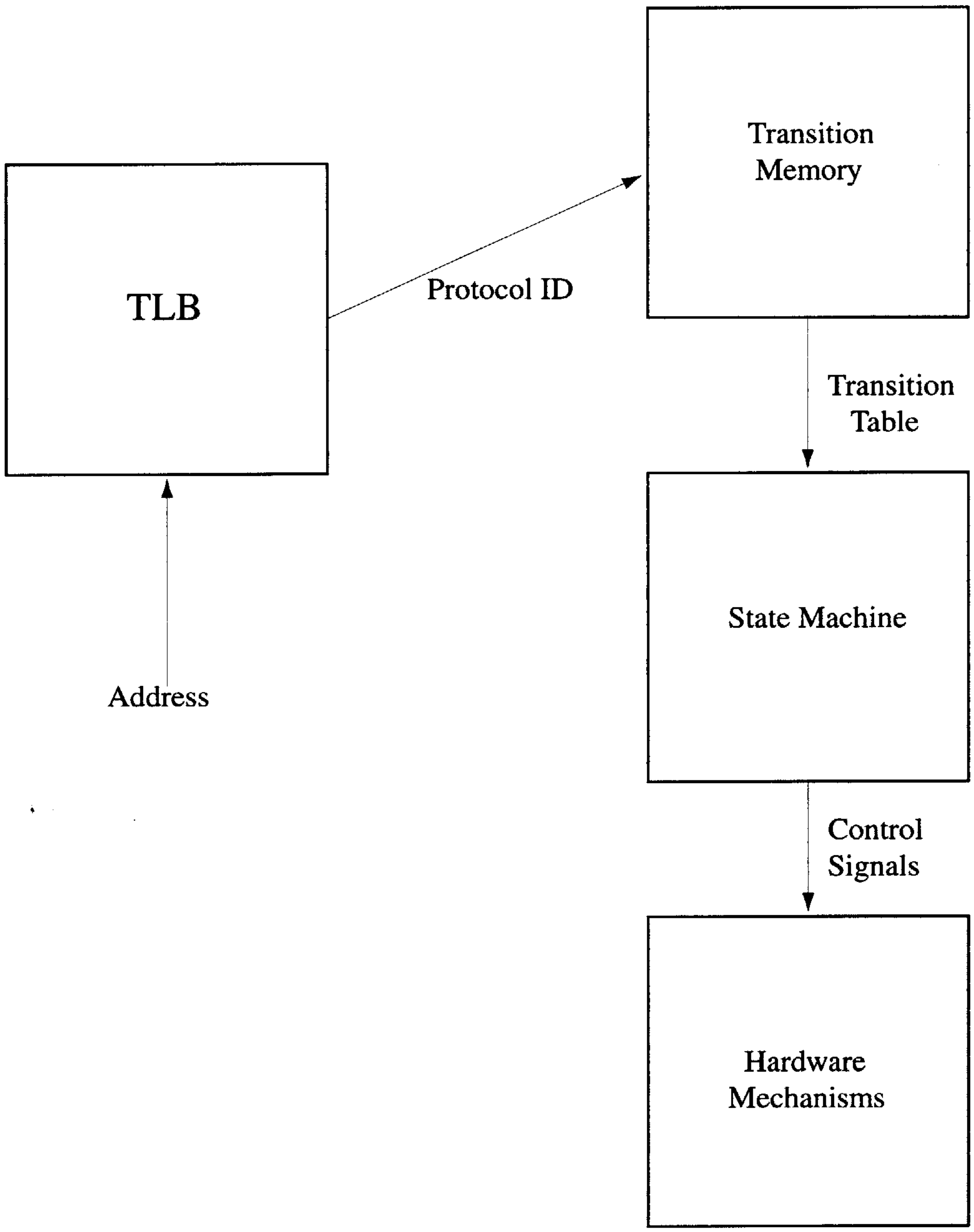


Figure 8.1: Flexible Shared-Memory System With Programmable State Machine

One promising technique for doing this would be to use a predictor to analyze the memory access pattern of a program and match the protocol to the needs of the application, creating an *adaptive* shared-memory system. This would greatly reduce the amount of programmer effort required to take advantage of flexibility in shared-memory systems. In addition, an adaptive shared-memory system would be able to react to changing access patterns by changing the shared-memory protocol used on a given block of data over the course of program execution, potentially providing better performance than could be achieved by compile-time analysis of the program to determine the best shared-memory protocol.

[27] showed that conventional prediction techniques are effective at predicting the behavior of shared-memory protocols to determine what the next coherence message received at a node would be, albeit at a high overhead. The predictor used in an adaptive shared-memory system would have to operate at a higher level to be effective, but could possibly achieve better results by making use of higher-level information. For example, if each sharing node were to track the number of times a remote block is accessed between installation and eviction (possibly using a small saturating counter), and were to send this information back to the home node of the block along with the acknowledgment that the eviction has been completed, the home node could use this information to decide whether an update- or invalidation-based protocol would be more effective for the block. Similarly, it might be possible to detect access patterns which perform poorly on conventional shared-memory protocols, such as producer-consumer synchronization, and select a protocol optimized for the access pattern.

Interfacing adaptive protocol-selection hardware with a flexible shared-memory controller is somewhat difficult, due to the difficulty of building hardware which can understand an arbitrary cache-coherence protocol well enough to decide when to use the protocol. One solution to this problem would be to reduce the flexibility of the shared-memory controller, designing the controller to implement a fixed set of protocols, but, as discussed earlier, this approach has a number of drawbacks. A better solution would be to assign a purpose to each of the entries in the memory which stores the state-transition tables in the flexible shared-memory controller, as shown in Figure 8.2.

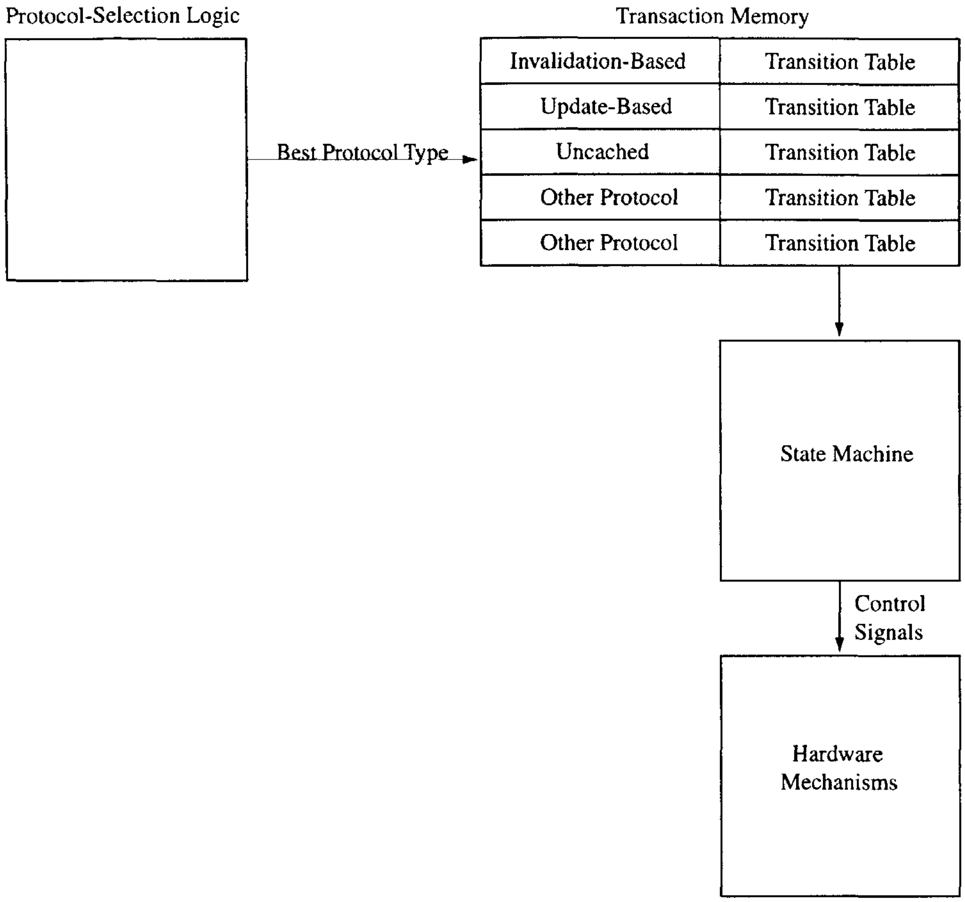


Figure 8.2: Adaptive Flexible Shared-Memory System

In this design, the protocol-selection logic would select the type of protocol for each region of data from the list of protocol types, and would use the type to index into the transition table memory to access the transition table which implements the specified protocol. This provides a wide spectrum of points on the effort/speedup curve for programmers to implement. The standard mode of operation would be to load a stock set of state-transition diagrams into the transition memory, and allow the adaptive logic to select among them. In cases where maximum performance is required, the programmer would have the option of designing protocols to the specific needs of the application and loading the state-transition diagrams for those protocols into the appropriate entries in the transition memory. Alternately, the adaptive logic can be effectively disabled by loading the same state-transition diagram into all of the entries in the transition memory.

An adaptive, flexible, shared-memory system would overcome many of the disadvantages of current shared-memory systems. The flexible shared-memory hardware would allow multiple protocols to be supported at hardware speeds, eliminating the performance penalties incurred by involving software in the resolution of remote requests, while the adaptive protocol-selection logic would automate the process of matching the protocol to the needs of the application, reducing programmer effort. In addition, the adaptive logic would have the ability to change the protocol being used on a given block of data dynamically over the course of program execution, potentially allowing such a system to achieve better performance than a system whose selection of shared-memory protocols is fixed for the entire execution time.

As semiconductor fabrication technology advances, flexible shared-memory systems will become increasingly attractive. Increases in transistor density will reduce the effective cost of the additional hardware required to provide adaptivity and flexibility by allowing more and more logic to be integrated onto a single chip. At the same time, the performance benefits achievable through flexibility will increase, because flexible systems improve performance by eliminating unnecessary communication. Since on-chip communication delays are increasing, and inter-chip communication delays are remaining mostly constant in spite of decreasing cycle times, each eliminated communication will have a greater and greater impact on program performance as fabrication technology advances.

This last factor is the strongest argument in favor of continued research into flexible shared-memory systems. Most architectural techniques, such as branch prediction or adding additional functional units to exploit instruction-level performance, reduce the amount of time required to perform computation. As fabrication technology improves, communication times will increase while computation times decrease, so the incremental performance available from additional mechanisms to reduce computation time will decrease, being limited by Amdahl's law. In contrast, because flexible shared-memory systems attack communication overheads, the potential speedup available through flexibility will increase over time, giving flexible systems a greater and greater advantage over conventional shared-memory systems.

Chapter 9

Conclusion

This thesis has described the M-Machine's combined hardware/software shared-memory system, which integrates mechanisms for shared memory into the MAP chip to improve the performance of shared-memory handlers. Four mechanisms have been implemented on the MAP: block status bits to allow small blocks of data to be transferred between nodes, a global translation lookaside buffer to allow determination of the home node of an address in hardware, a fast event system to detect remote accesses and invoke software handlers, and dedicated thread slots for handlers to eliminate context switch overhead. In combination with the MAP chip's multithreaded processor architecture, these mechanisms allow shared-memory handlers to efficiently share processor resources with user threads, improving processor utilization.

Using just the block status bits, the M-Machine is able to complete a remote memory access in 523 cycles, significantly faster than any contemporary software-only shared memory system. Adding either the GTLB or the MAP chip's multithreading scheme reduces the remote access time to 427 and 433 cycles, respectively, and using all of the M-Machine's mechanisms allows remote accesses to be completed in 336 cycles.

Two programs were run using a simple cache-coherence protocol to determine the impact of the M-Machine's mechanisms for shared memory on application performance: a 1,024-point FFT and an 8x8x8 multigrid computation. These programs display very different memory access patterns. In FFT, the remote memory requests are relatively evenly distributed across the nodes and utilization of the shared-memory handler threads is low, making remote memory latency the dominant factor in program performance. In multigrid, however, the vast majority of the remote memory requests go to one node, due to the small size of the matrices used in the computation. This causes the occupancy of the request handler thread on that node to be very high, making handler thread occupancy the most important factor in this program's performance.

One interesting result from the program-level experiments was the number of conflict misses in the LTLB during execution of shared-memory programs. Executing shared-memory handlers on the same processor as user programs increases the demands on the translation hardware because the addresses referenced by the shared-memory handlers are independent of the addresses referenced by the application. For both of the applications studied, this resulted in high LTLB miss rates in the MAP chip's 64-entry, direct-mapped LTLB. More importantly, the variance in miss rates between versions of the cache-coherence protocol masked the performance impact of the MAP chip's mechanisms, as shared-memory handlers which use fewer of the mechanisms reference virtual addresses less frequently, reducing the number of conflict misses.

Using a 128-entry, two-way set-associative LTLB substantially reduced the LTLB miss rate for both applications, allowing analysis of the performance impact of the M-Machine's mechanisms for shared memory. On FFT, the program-level impact of each of the mechanisms correlated well with the mechanism's impact on remote access times, with the multithreading scheme giving improvements of up to 5%, the GTLB improving performance by up to 4%, and the combination of the two mechanisms giving a 9% performance boost.

On multigrid, however, the performance impact of the mechanisms varied substantially depending on which part of the shared-memory protocol they accelerate. The multithreading scheme improved performance by up to 20%, an unsurprising result given that multigrid spends much more time waiting for memory than FFT. However, the GTLB gave a maximum speedup of 2%, and reduced performance in one case, even though the remote access time when using just the GTLB is very close to the remote access time when using just the multithreading scheme. The explanation for this behavior comes from the fact that multigrid's performance is dominated by the occupancy of the request handler thread on the hot-spot node. The MAP chip's multithreading scheme improves the performance of all of the shared-memory handler threads by eliminating context switches at the start and end of each handler, which reduces the occupancy of the request handler thread slot. The GTLB, however, improves the performance of the event handler thread by eliminating the need to determine the home node of an address in software. Since the event handler thread is not the bottleneck for this application, improving its performance has only a small impact on overall execution time.

Analysis of the shared-memory system suggested three additional mechanisms to improve performance: automatic configuration space address generation, instructions that ignore block status bits, and transaction and lock buffers. Automatic generation of configuration space addresses improves performance by eliminating the need to generate the configuration space address to complete a remote memory operation in software, while instructions which ignore block status bits eliminate the need to use physical addresses to reference blocks of data during block eviction and installation, reducing the number of off-chip memory references required during the resolution of a remote memory reference. Transaction and lock buffers track pending remote memory references in hardware, eliminating the need to maintain a pending operation data structure.

Automatic configuration space address generation reduces remote memory access time from 338 to 331 cycles, while instructions that ignore block status bits reduce the remote access time to 318 cycles. Combining these two mechanisms gives a remote access time of 313 cycles. Adding transaction and lock buffers to the other two mechanisms reduces the remote access time to 229 cycles.

For the application programs studied, transaction and lock buffers were found to give substantial performance improvements, reducing execution time by up to 16%. Instructions which ignore block status bits gave speedups of up to 6%. Automatic generation of configuration space addresses improved performance by up to 1% on FFT, but reduced performance slightly on some of the multigrid experiments. Again, the explanation for this behavior comes from the part of the shared-memory protocol affected by the mechanisms. Automatic generation of configuration space addresses reduces the execution time of the event handler, which is responsible for configuration space address generation in the base protocol. On a latency-dominated application like FFT, this improves performance by reducing the remote reference latency. On an occupancy-dominated benchmark like multigrid, increasing the performance of the event handler can reduce program performance by increasing the rate at which requests arrive at the hot-spot node, increasing the number of requests which have to be bounced back to their senders to be retried later.

This thesis has shown that the addition of hardware mechanisms for shared memory to a multithreaded processor provides both good shared-memory performance and flexibility by providing hardware acceleration for tasks common to many shared-memory protocols while relying on software to perform tasks which vary between protocols. The hardware cost of these mechanisms is much smaller than that of flexible shared-memory systems which rely on dedicated co-processors to execute shared-memory handlers. The mechanisms implemented on the MAP require less than 3KB of SRAM and the register files for the dedicated handler thread slots to implement, while the new mechanisms proposed in this thesis require approximately 6KB of SRAM in addition to the area occupied by the MAP's mechanisms.

Appendix A

Coherence Protocol Specification

Chapter 4 described the M-Machine's mechanisms for shared memory and how they are used to implement software shared memory. This appendix describes in more detail how the M-Machine's mechanisms were used to implement the three-hop, invalidation-based protocol used in this thesis.

A.1 Threads Used in the Protocol

Figure A.1 shows the assignment of threads to hardware thread slots on the MAP when the cache coherence protocol is being used. Threads shown in italics are involved in the coherence protocol. As described in Chapter 4, the event handler thread executes the event handlers which respond to remote memory requests and send request messages. The priority 0 (P0) message handler thread processes remote memory and invalidation requests, while the priority 1 (P1) message handler thread processes replies from remote requests, acknowledgments of invalidations, and bounced messages.

Bounced messages occur when the P0 message handler receives a request that cannot handle, such as a request for a block which is in the process of being invalidated or an invalidation request for a block which has not yet been installed on the node. When this happens, the P0 message handler creates a P1 message containing the original message and sends it back to the originating node to be resent later. When the P1 message handler on the originating node receives the bounce message, it places it in a software queue, to be resent by the bounce proxy thread.

V-Thread	Cluster 0	Cluster 1	Cluster 2
0	User	User	User
1	User	User	User
2	*Event Handler*	*Evict Proxy*	*Bounce Proxy*
3	Exception	Exception	Exception
4	LTLB Miss	*P0 Message*	*P1 Message*

Figure A.1: Thread Slot assignments for the Cache Coherence Protocol

A.1.1 Deadlock Avoidance

In most cases, the two message priorities on the MAP allow deadlock avoidance through a simple request-reply protocol: request messages go out on priority 0, and reply messages come back on priority 1. Priority 0 message handlers are only allowed to send priority 1 messages, and priority 1 message handlers are not allowed to send any messages at all. However, there are two cases in the protocol where this approach does not suffice: invalidations and bounced messages. To avoid deadlock in these situations, the protocol makes use of two proxy threads: the evict proxy, and the bounce queue.

The evict proxy thread handles the sending of invalidation messages, avoiding the deadlock situation that would occur if the priority 0 message handler sent priority 0 invalidation messages itself. When a request handler running in the priority 0 message handler slot determines that a block needs to be invalidated in order to resolve a remote memory request, it checks to see if the evict proxy thread is currently busy. If the evict proxy is busy, the request handler bounces the original message back to the requesting node to be retried later. If not, it uses the configuration space to write into the evict proxy's register file the address to be evicted, a pointer to the page sharing entry which describes the set of nodes that have copies of the block, and the data required to resolve the original request.

The evict proxy then sends the appropriate invalidation messages to the sharing node and terminates. Since the evict proxy thread does not reside in one of the dedicated message handler thread slots, there is no danger of deadlock when it sends messages.

The bounce proxy is responsible for re-sending messages that have been bounced back to their sending node because their destination node was unable to resolve them. When a bounced message arrives at the P1 message handler on the originating node, the P1 handler thread reads the contents of the bounced message out of the message queue, and places the original message into a software queue. Then, the P1 message handler writes into the register file of the bounce proxy thread, telling it that a message has been placed in the queue for it to resend. The bounce proxy thread then extracts the original message from the queue, and re-sends it to the destination node, avoiding deadlock because it does not run in a message handler thread slot.

A.2 Data Structures

The two main data structures used by the protocol are the pending operation structure and the directory. The pending operation structure tracks remote memory requests which have been processed by the event handler but not yet resolved, while the directory tracks the set of nodes which have copies of blocks that the node is the home node for. Each of these data structures is stored in the physically-addressed local memory of each node, to ensure that the shared-memory handlers never reference remote data, which could deadlock the protocol.

A.2.1 The Pending Operation Structure

The pending operation structure is an array of *remote page records*, with one remote page record per physical page used on the local node. Remote page records are twenty words long, and are created by the LTLB miss handler thread during the resolution of the first reference to a given page of memory. The array of remote page records is accessed by open hashing, using the virtual address of the page to generate the hash key. In the run-time system used for this thesis, pages of physical memory are never deallocated once they have been mapped onto virtual pages, and therefore it is not necessary to delete remote page records because the physical page they describe has been allocated to another virtual page. In a commercial shared-memory system based on the M-Machine, it would be necessary to develop some mechanism for managing the pending operation structure in the face of re-assignment of physical pages.

The first two words of each remote page record contain the virtual and physical addresses of the page. The virtual address of the page is required in order to determine when the correct pending operation record has been located, while the physical address is needed for block eviction and installation, as was discussed in Chapter 6. The third word is used as a bit vector, assigning one bit to each of the 64 blocks in the page mapped by the record, and tracks the set of blocks for which a request for a writable copy of the block has been sent but not received. The fourth word tracks the set of blocks for which read-only requests have been sent, in the same fashion.

These bit vectors of pending requests are used to prevent a node from having multiple outstanding requests for the same block of data. When the event handler processes an event caused by a remote data reference, it checks the bit vectors of the appropriate remote page record, and does not send an additional request if an appropriate (writable for store operations, read-only or writable for loads) request has already been sent. This eliminates redundant requests, simplifying the handlers and reducing network congestion. The synchronization bit on the first bit vector word is also used as a lock on the remote page record, to prevent multiple handlers from modifying the same remote page record simultaneously.

Word	Contents
0	Virtual Address
1	Physical Address
2	Bit-vector of lines with pending write operations
3	Bit-vector of lines with pending read operations
4	Offset to pending operation records for lines 0-3
5	Offset to pending operation records for lines 4-7
6	Offset to pending operation records for lines 8-11
7	Offset to pending operation records for lines 12-15
8	Offset to pending operation records for lines 16-19
9	Offset to pending operation records for lines 20-23
10	Offset to pending operation records for lines 24-27
11	Offset to pending operation records for lines 28-31
12	Offset to pending operation records for lines 32-35
13	Offset to pending operation records for lines 36-39
14	Offset to pending operation records for lines 40-43
15	Offset to pending operation records for lines 44-47
16	Offset to pending operation records for lines 48-51
17	Offset to pending operation records for lines 52-55
18	Offset to pending operation records for lines 56-59
19	Offset to pending operation records for lines 60-63

Figure A.1: Remote Page Record Format

The remaining words of the structure contain offsets that are used to locate the tails of the linked list of pending operations that are waiting for data contained in each block in the page. As a compromise between speed and storage required, the remote page record does not store pointers to the tails of each list. Instead, four 16-bit offsets into the array of pending operation records are stored in each word of the structure, allowing the tail of each list to be found quickly, but substantially reducing the storage required.

Pending operation records, as shown in Figure A.2, record the information necessary to resolve a single remote memory operation. Four words of data are required to resolve an operation: the referenced address, the operation's data, the configuration space address which will be used to resolve the operation, and the address of the routine which will resolve the operation. The configuration space address and address of the resolution routine are stored in the pending operation structure in order to move as much work as possible into the event handler for load-balancing purposes. Either the event handler or the reply handler could be responsible for computing these addresses, but handling these tasks in the event handler increases the time to create a remote memory request rather than the time to resolve one, making it harder to create requests faster than they can be resolved.

Word	Contents
0	Operation Word Address
1	Offset to Next Record
2	Operation Type (Pointer to Code to Resolve Operation)
3	Operation Data
4	Configuration Space Pointer to Operation Destination
5	Unused
6	ACKs Required
7	Unused

Figure A.2: Pending operation Record Format

The first word of the pending operation record is used to implement a linked list of records which describe the operations waiting for a given block of memory. When the first request for a given block of memory arrives, the event handler creates a record for it, places the offset of that record in the appropriate place in the remote page record for the page, and sends a pointer to the record to the home node as part of the request message.

This pointer is returned as part of the reply message, allowing the reply handler to access the head of the list even though no pointer to the head of the list is kept in the pending operation structure. If additional references to the same block occur before the reply arrives, the event handler adds them to the linked list by writing the offset to the new pending operation record into the next field of the tail record of the list and then writing the offset of the new tail into the remote page record. This allows additional pending operation records to be added to the list in constant time without increasing the time required to locate the head of the list.

An array of free pending operation records is allocated on each node at system startup, and linked together by the startup code into a free record list. The offsets of the head and tail of this list are stored in the system data segment, making it easy to put pending operation records back on the free list once their operation has been resolved. If the free list of pending operation records on a node ever becomes empty, the event handler is prohibited from sending more request messages until some of the outstanding requests have been resolved. Since the event handler is not involved in resolving requests, having the event handler wait in this manner does not create a potential deadlock.

A.2.2 The Directory

Each node maintains a *directory* which tracks the set of nodes which have cached copies of data that it is the home node for. The directory consists of an array of twenty-word records, one record for each physical page on the node. This array is accessed by open hashing on the address of the data being requested, using the same hash algorithm as the pending operation structure to reduce hash computation time.

Each entry in the directory consists of twenty words. The first two words contain the virtual and physical addresses of the page mapped by the entry, while the third contains a bit vector describing which blocks in the page are currently being invalidated. Word four is currently unused, while words 5-20 contain offsets into the array of page sharing records for the page sharing records for each of the blocks in the page, using the same scheme as the pending operation structure.

Each offset is 16 bits long, which allows a maximum of 64K page sharing records. Since each page sharing record describes an 8-word line, this scheme allows up to 512K words, or 4MB of data, to be mapped remotely, less if some lines are widely shared and thus require more than one page sharing record. This scheme allows up to half of the 8MB of memory that the M-Machine provides on each node to be shared while limiting the space required for the directory by packing four offsets into each data word. If the external memory on each node were larger, this data structure could be modified to use 32- or 64-bit offsets, allowing more data from each node to be shared without substantially changing the performance of the cache-coherence scheme, although it would increase the percentage of each node's memory dedicated to the directory.

Word	Contents
0	Virtual Address of Page
1	Physical address of page
2	Bit Vector of Invalidations in Progress
3	Unused
4	Offsets to Page Sharing Records for lines 0-3
5	Offsets to Page Sharing Records for lines 4-7
6	Offsets to Page Sharing Records for lines 8-11
7	Offsets to Page Sharing Records for lines 12-15
8	Offsets to Page Sharing Records for lines 16-19
9	Offsets to Page Sharing Records for lines 20-23
10	Offsets to Page Sharing Records for lines 24-27
11	Offsets to Page Sharing Records for lines 28-31
12	Offsets to Page Sharing Records for lines 32-35
13	Offsets to Page Sharing Records for lines 36-39
14	Offsets to Page Sharing Records for lines 40-43
15	Offsets to Page Sharing Records for lines 44-47
16	Offsets to Page Sharing Records for lines 48-51
17	Offsets to Page Sharing Records for lines 52-55
18	Offsets to Page Sharing Records for lines 56-59
19	Offsets to Page Sharing Records for lines 60-63

Figure A.3: Directory Entry Format

A.2.3 Page Sharing Records

Page sharing records, as shown in Figure A.4, track the set of nodes that have copies of a block of data. The low 16 bits of the first word contain a count of the number of nodes which share the page. The next 16 bits contain an offset into the array of page sharing records at which the next page sharing record for the line lies. Words 2-4 contain the node ID's of up to twelve nodes which have copies of the block. If more than twelve nodes share a line, additional page sharing records may be chained together to form a linked list. The count field of the first page sharing record always contains the total number of nodes which share the page, allowing the handler to easily determine where the next free slot to record the ID of a sharing node is. Since the count field is the same length as a node ID on the M-Machine (16 bits), there is no possibility that the number of nodes which share a line will overflow the count field.

Word	Bits 63-48	Bits 47-32	Bits 31-16	Bits 15-0
0	Unused	Unused	Next	Count
1	Node ID 3	Node ID 2	Node ID 1	Node ID 0
2	Node ID 7	Node ID 6	Node ID 5	Node ID 4
3	Node ID 11	Node ID 10	Node ID 9	Node ID 8

Figure A.4: Page sharing record format

An array of page sharing records is allocated at boot time, and formed into a linked list by setting the next fields of each record in the array. The offsets into the array for the head and tail of the free page sharing record list are kept in the system data segment. The protocol implemented for this thesis does not handle the case where the list of free page sharing records becomes empty. In a commercial system, it would be necessary to handle this case by allocating more records, evicting blocks of data to free up records, or some combination of the two.

A.3 Messages Used to Implement the Protocol

A.3.1 RREQ and WREQ

RREQ and WREQ are the basic remote memory request messages, requesting read-only and writable copies of a block, respectively. These messages are 4 words long, and contain the address being requested, the hash offsets for the page containing the address (to avoid having to re-compute them at the destination node), and pointers to the remote page record and pending operation record for the data being requested, to improve the performance of the reply handler. RREQ and WREQ messages are sent on priority 0.

A.3.2 REQ_ACK_DATA_W and REQ_ACK_DATA_R

These messages are the basic reply messages, and are sent from the home node to the requesting node on priority 1 when it is not necessary to invalidate a block in order to satisfy a remote memory request. REQ_ACK_DATA_W messages are sent in response to requests for writable data, and cause the block to be installed on the home node in a writable state, while REQ_ACK _DATA_R messages cause the block to be installed in a read-only state. Each of these messages is 11 words long, containing the address of the referenced block, pointers to the remote page record and pending operation record for the block, and the eight words of data in the block.

A.3.3 INV_W and INV_R

INV_W and INV_R messages are sent on priority 0 by the home node of a block to each sharing node when it is necessary to invalidate a block. INV_W and INV_R messages are 6 words long, containing the address of the block, pointers to the remote page record and pending operation record on the requesting node, a pointer to the directory entry for the page on the home node, the hash offsets for the address, and a word which contains the node ID of the requesting node, the number of sharing nodes, and the status that the block should be set to on the requesting node.

INV_W messages expect the block to be in a writable state on the sharing node, while INV_R messages expect the block to be in a read-only state, in order to prevent errors which could otherwise result if an invalidation message arrived on a node ahead of the reply message which contains the block being invalidated. If this happened, the reply message could install the block after the invalidation message had completed, leading to the node having a copy of the block in spite of the fact that the home node believes that the sharing node's copy has been evicted.

To prevent this, the home node sends either an INV_R or INV_W message, depending on whether it believes that the sharing node has a read-only or a writable copy of the block. If the state of the block on the sharing node does not match the expectations of the home node, the sharing node bounces the invalidation message back to the home node, and this process repeats until the reply message arrives on the sharing node and sets the block to the status expected by the home node.

A.3.4 INV_ACK and INV_DATA

When a sharing node finishes processing an invalidation request, it sends two messages on priority 1: an INV_DATA message to the requesting node, and an INV_ACK message to the home node. INV_DATA messages are 12 words long, and contain the address of the block, the contents of the block, pointers to the remote page record and pending operation record for the block, and a word which contains the number of nodes which had copies of the block before the eviction started and the status which the block should be set to once all the data messages have arrived on the sharing node. INV_ACK messages use the same format, but the pointer to the remote page record has been replaced by a pointer to the directory entry for the block on the home node. This is somewhat inefficient, as it results in the transfer of unnecessary data across the network, but allows re-use of most of the INV_DATA message in the INV_ACK message, reducing message composition time.

A.3.5 BOUNCE

When the P0 message handler on a node cannot resolve a request immediately, it sends the request message back to the originating node using a BOUNCE message. BOUNCE messages are sent on priority 1, and are of variable length, containing the node ID of the destination node and the original message.

References

[1] Anant Agarwal *et al.* The MIT Alewife machine: architecture and performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2-13, June 1995.

[2] Anant Agarwal *et al.* Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48-61, June 1993

[3] David I. August *et al.* Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227-237, June 1998.

[4] Ricardo Bianchini. Application performance on the Alewife multiprocessor. Alewife Systems Memo #43, Laboratory for Computer Science, Massachusetts Institute of Technology, 1994.

[5] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS VI), pages 319-327, October 1994.

[6] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: a scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS IV), April 1991.

[7] Satish Chandra, Brad Richards, and James R. Larus. Teapot: language support for writing memory coherence protocols. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[8] Andrew Chang, William J. Dally, Stephen W. Keckler, Nicholas P. Carter, Whay S. Lee. The effects of explicitly parallel mechanisms on the Multi-ALU processor cluster pipeline. In *Proceedings of the 1998 International Conference on Computer Design* (ICCD '98), October 1998.

[9] Andrew Chang. VLSI datapath choices: cell-based vs. full-custom. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1998.

[10] R. S. Fabry. Capability-based addressing. In *Communications of the ACM*, July 1974, pages 403-412.

[11] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 146-156, Ann Arbor, MI, December 1995

[12] Yevgeny Gurevich. The M-Machine operating system. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1995.

[13] Mark Heinrich *et al.* The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS VI), pages 274-285, October 1994

[14] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS VI), pages 38-50, October 1994

[15] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364-373, 1990.

[16] Stephen W. Keckler. Fast thread communication and synchronization mechanisms for a scalable single chip multiprocessor. Ph. D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1998.

[17] Stephen W. Keckler and William J. Dally. Processor Coupling: integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 202-213, May 1992.

[18] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting fine-grain thread-level parallelism on the MIT Multi-ALU processor. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[19] Jeffery Kuskin *et al.* The stanford FLASH multiprocessor. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 302-313. June 1994

[20] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th Annual Symposium on Computer Architecture*, pages 241-251. June 1997

[21] Whay S. Lee, William J. Dally, Stephen W. Keckler, Nicholas P. Carter, and Andrew Chang. Efficient, protected message interface in the MIT M-Machine. In *IEEE Computer Special Issue on Design Challenges for High-Performance Network Interfaces*, November 1998.

[22] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148-159, 1990.

[23] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92-103, May 1992.

[24] Kai Li. IVY: a shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing*, pages 94-101, 1988.

[25] P. G. Lowney, S. G. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnel, and J. C. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51-142, May 1993.

[26] Daniel Maskit. Software register synchronization for super-scalar processors with partitioned register files. Ph.D. thesis, California Institute of Technology, 1997.

[27] Shubhendu S. Mukherjee and Mark D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 179-190, June 1998.

[28] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: an architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224-235, San Diego, California, May 1993

[29] 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 325-337, April 1994.

[30] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, pages 34-43, May 1996

[31] M. Satyanarayanan, Commercial multiprocessing systems. IEEE Computer, May 1980.

[32] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grained shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VII), pages 174-185, October 1996.

[33] Ioannis Schoinas *et al.* Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS VI), pages 297-306, October 1994.

[34] Semiconductor Industry Association. The national technology roadmap for semiconductors, 1997

[35] Andrew Shultz. Advances in the M-Machine runtime system. Master's thesis, Massachusettts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1997.

[36] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cachemere-2L: software coherent shared memory on a clustered remote-write network. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[37] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Sattherwaite, Jr. Firefly: a multiprocessor workstation. Technical Report 23, Digital Systems Research Center, December 1987.

[38] David A. Wood, Susan Eggers, and Garth Gibson. SPUR memory system architecture. Technical Report UCB/CSD 87/395, University of California, Berkeley, December 1987.

[39] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: a multigrain shared memory system. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, pages 44-55, May 1996