Appears in the International Symposium on High-Performance Computing, Tokyo, Japan, October 2000

Processor Mechanisms for Software Shared Memory

Nicholas P. Carter[1], William J. Dally[2], Whay S. Lee[3], Stephen W. Keckler[4], and Andrew Chang[2]

[1] Coordinated Science Laboratory, University of Illinois at Urbana-Champaign
{npcarter@crhc.uiuc.edu}
[2] Computer Systems Laboratory, Stanford University
{billd@cva.stanford.edu, achang@cva.stanford.edu}
[3] Sun Microsystems
{Whay.Lee@EBay.Sun.COM}
[4] Department of Computer Sciences, The University of Texas at Austin
{skeckler@cs.utexas.edu}

Abstract. The M-Machine's combined hardware-software shared-memory system provides significantly lower remote memory latencies than software DSM systems while retaining the flexibility of software DSM. This system is based around four hardware mechanisms for shared memory: status bits on individual memory blocks, hardware translation of memory addresses to home processors, fast detection of remote accesses, and dedicated thread slots for shared-memory handlers. These mechanisms have been implemented on the MAP processor, and allow remote memory references to be completed in as little as 336 cycles at low hardware cost.

1 Introduction

Distributed Shared-Memory (DSM) systems use a variety of methods to implement shared-memory communication between processors. Some designers provide substantial hardware support for shared memory, such as hardware protocol engines [1] or dedicated co-processors [10]. Other systems rely completely on software to implement shared memory [13]. Hardware protocol engines can give very low remote memory latencies, but increase the complexity and hardware cost of the system. In addition, they restrict the system to one shared-memory protocol, limiting performance on applications whose communications patterns do not match the assumptions of the shared-memory protocol. Software-based shared-memory is very flexible, and does not increase the cost of the system, but tends to have relatively poor performance due to the overheads imposed by conventional networks and virtual memory systems. Providing co-processors to execute shared-memory handlers gives both flexibility and speed, but the hardware cost of this approach is substantial.

In this paper, we present an alternative approach to implementing DSM systems, based around four hardware mechanisms for shared memory that are integrated into the processor itself, substantially improving shared-memory performance at low hardware cost while retaining the flexibility of software-based approaches. Shared-memory protocols share several common features, which can be exploited in the design of hardware mechanisms to accelerate shared memory. They must be able to detect references to remote memory, determine where the request for the remote memory must be sent, transfer data back to the requesting processor, and complete operations which are waiting for the data. In addition, it is desirable that an architecture support transfers of small blocks of data between processors to reduce false sharing (unnecessary remote memory operations caused by placing two or more unrelated data objects within a block of memory that the shared-memory system treats as an atomic unit), and that the architecture allow user programs to continue executing during remote memory references.

Based on these requirements, we have designed four hardware mechanisms for shared memory, which have been implemented as part of the MIT M-Machine project [5]. Block status bits on each eight-word block of memory allow individual blocks to be transferred between processors. A fast event system detects remote memory references and invokes software handlers in as little as 10 cycles. A Global Translation Lookaside Buffer caches translations between virtual addresses and their home processors. Dedicated thread slots for software handlers eliminate context switch overhead when starting handlers, and allow user programs to execute in parallel with shared-memory handlers.

Using all of our mechanisms allows system software to complete a remote memory reference in 336 cycles on the M-Machine, almost 20x faster than most software-only shared-memory systems, and only 2.5x slower than current-generation hardware shared-memory systems such as the SGI Origin 2000 [11]. On applications, we achieve a 9% performance improvement on a latency-bound FFT computation and a 30% improvement on an occupancy-bound multigrid computation, as compared to the performance of our system without these hardware mechanisms [4].

The remainder of this paper begins with a brief overview of the MIT M-Machine, followed by a description of the mechanisms and their use in implementing shared memory. We continue with an analysis of the performance impact of our mechanisms for shared memory, followed by a discussion of related work and some future research directions.

2 The M-Machine

The M-Machine Multicomputer is an experimental multicomputer that we have designed at M.I.T. and Stanford to explore architectural techniques to take advantage of improvements in silicon fabrication technology. An M-Machine consists of a two-dimensional array of processing nodes, each of which consists of a custom Multi-ALU processor (MAP) and five SDRAM chips. The MAP chip has been fabricated in a 0.5-micron process, and work is ongoing as of July, 2000 on a prototype M-Machine.

As shown in Figure 1, each MAP chip contains three processor clusters [8], two cache memory banks, and a network subsystem. Each of the clusters acts as an independent, multithreaded processor. The instruction issue logic in each cluster implements zero-overhead multithreading between the five active threads in the cluster, selecting an instruction to issue each cycle based on operand and resource availability. Threads running in the same thread slot (hardware registers which hold program state) on each of the clusters are assumed to be part of the same job, and may use the cluster switch to write into each other's register files. Memory addresses are interleaved between the two cache banks, allowing two memory operations to be completed each cycle.

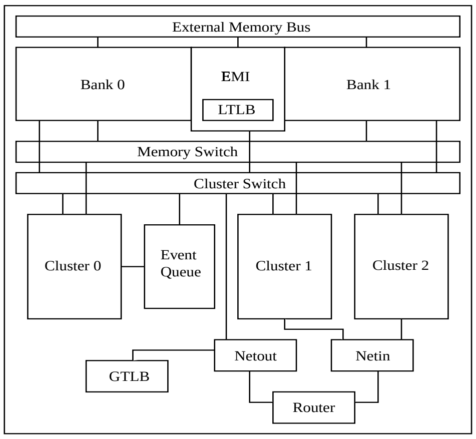

Figure 1: MAP Chip Block Diagram

The MAP chip's architecture allows it to exploit parallelism at multiple granularities. Each cluster acts as a 2- or 3-wide LIW processor[1], allowing fine-grained instruction-level parallelism to be exploited within a cluster. The MAP chip's inter-cluster communication mechanisms [9] provide low-latency communication between threads running on different clusters, making it feasible to exploit medium-grained parallelism within a MAP chip. Finally, the on-chip network hardware [12] provides fast, user-level messaging between processors in an M-Machine, allowing coarser-grained parallelism to be exploited across multiple processors.

Shared memory is implemented on the M-Machine using a combination of hardware and software. Hardware detects remote memory operations and passes them to software to resolve. Remote memory references are enqueued in software while their data is being obtained, and are then resolved by the shared-memory handlers. Completing pending operations in software is made easier by the MAP chip's *configuration space*, a mechanism which maps all of the register state of the chip into an address space that can be accessed using normal load and store operations, relying on the Guarded Pointers [3] protection scheme to prevent unauthorized programs from modifying other program's register states. This allows the system software to write the result of each pending operation directly into its original destination register, making software resolution of remote memory requests transparent to user programs.

1. The original design called for each cluster to have 3 ALUs: one integer, one memory, and one floating-point. Chip-space constraints forced the removal of the FP ALUs from two of the clusters during implementation.

3 Mechanisms for Shared Memory

We have designed four hardware mechanisms for software shared memory: block status bits, a fast event system, a global translation lookaside buffer, and dedicated thread slots for shared-memory handlers. Together, these mechanisms implement several important sub-tasks of most shared-memory protocols in hardware, freeing the software handlers to implement higher-level DSM policies. Block status bits allow 8-word blocks of data to be transferred between processors, reducing false sharing. The event system detects remote references and invokes software handlers to resolve them. The Global Translation Lookaside Buffer provides a flexible mapping of addresses to home processors, allowing data to be mapped for maximum locality. Dedicated thread slots for software handlers eliminate context switch overheads when invoking software handlers, reducing the remote reference time.

3.1 Block Status Bits

The block status bits allow small blocks of data to be transferred between processors by associating two bits of state with each 8-word block of data on a node. These bits encode the states invalid, read-only, read-write, and dirty, and the hardware enforces the permissions represented by these states. Block status bits eliminate much of the false sharing which occurs in software-only shared memory systems because conventional virtual memory systems are unable to record presence information on units of data smaller than a page. They are stored in the page table and copied into the local TLB (LTLB) and the cache when a block is referenced, allowing remote data to be stored at all levels in the memory hierarchy.

During a memory reference, the memory system checks the block status bits of the referenced address to determine if the operation is allowed. This check is done in parallel with hit/miss determination in the cache or LTLB and therefore does not impact memory latency. If the block status bits allow the operation, it is completed in hardware. Otherwise, the event system starts a software handler to resolve the operation. Implementing block status bits requires 1KB of SRAM in the LTLB, and 0.25 KB of SRAM in the caches.

3.2 The Event System

The event system is responsible for invoking software handlers in response to remote memory accesses and other events which require intervention by system software. When the hardware detects a situation which requires software intervention, such as a remote memory reference, it places an event record describing the situation in a 128-word hardware *event queue*. It then discards the original operation, allowing programs to continue to execute while the event is resolved. A dedicated event handler thread processes the event records and resolves events.

The head of the event queue is mapped onto a register in the event handler's register file. This speeds up queue accesses and provides a low-overhead mechanism for blocking the event handler when there are no pending events. If the event queue is empty, the register for the head of the queue is marked empty by the scoreboard logic, preventing instructions that read the register from issuing. When the event handler tries to read the head of the event queue to begin processing the next event, the instruction stalls while the queue is empty, but issues as soon as an event record is placed in the queue, allowing the event handler to respond quickly to events without consuming execution cycles that could be used by other threads in polling.

3.3 Dedicated Thread Slots for Shared-Memory Handlers

The M-Machine takes advantage of the MAP chip's multithreaded architecture to eliminate context switch overhead from software handlers by dedicating a set of thread slots to software handler threads. Since the handler threads are always resident in their thread slots, there is no need to perform a context switch when a handler executes, in contrast to single-threaded implementations of software shared memory.

Thread	Cluster 0	Cluster 1	Cluster 2
0	User	User	User
1	User	User	User
2	Exception	Exception	Exception
3	*Event Handler*	*Evict Proxy*	*Bounce Proxy*
4	LTLB Miss Handler	*Request Handler (P0 Message Handler)*	*Reply Handler (P1 Message Handler)*

Figure 2: Thread Slot Assignments on the MAP Chip

Figure 2 shows the thread slot assignments on the MAP chip when shared-memory programs are being run. Threads which are involved in implementing shared memory are shown in italics, while thread slots which include special hardware are underlined. The event and message handlers process the events which occur when a remote memory reference is made as well as the various messages which are used to implement the shared-memory protocol. In addition, two thread slots are used for *proxy* threads, which are used to break potential deadlock situations that occur when the shared-memory system needs to send three sequential messages over the MAP chip's two network priorities.

Allocating thread slots to software handlers significantly improves the M-Machine's remote access time, and simplifies the design of the software handlers because a handler is never interrupted to allow another handler to execute. The main incremental cost of dedicating thread slots to handlers is the 1.25KB of storage required to hold the register files of the dedicated threads, since the substantial hardware complexity incurred by multithreading is required by the MAP chip's base architecture.

3.4 The Global Translation Lookaside Buffer

Determining how data will be mapped across the processors in a DSM is an important part of program implementation. Mapping data for maximum locality can substantially reduce the number of remote references made by a program, and thus improve performance. However, providing flexibility in address mapping increases the latency of software shared memory, as the shared-memory handlers must translate each remote reference to find the home processor of the reference, creating another area where a small amount of hardware support can significantly improve performance.

The Global Translation Lookaside Buffer (GTLB) acts as a cache for translations between virtual addresses and their home processors, similar to the way a normal TLB caches translations between virtual and physical addresses. The format of a GTLB entry (Figure 3) allows each entry to map a variable-sized group of pages across variable-sized regions of the machine. The data mapped by a GTLB entry is specified by a base address and a size field, which specifies the number of pages in the page group. The region of the machine that the page group is mapped across is specified by its start node, the X- and Y-extents of the region (in the 2-D network), and the number of contiguous pages mapped per node. All fields except the base address and the start node are logarithmically encoded to reduce space.

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

Figure 3: GTLB Entry Format

The GTLB allows substantial flexibility in mapping addresses. For example, Figure 4 shows the three ways in which 16 pages of data can be mapped across a 2x2 block of processors. Note that changing the value of the start field allows the pages to be translated across the machine, facilitating space sharing of a multiprocessor.

The GTLB's entry format allows it to be implemented in very little hardware. The MAP chip implements a 4-entry GTLB due to space constraints (a 16-entry GTLB was specified in the initial design), which requires 64 bytes of content-addressable memory. For the experiments run for this paper, only two GTLB entries were required -- one to map the code segment locally on each processor, and one to map the data segment across the entire machine.

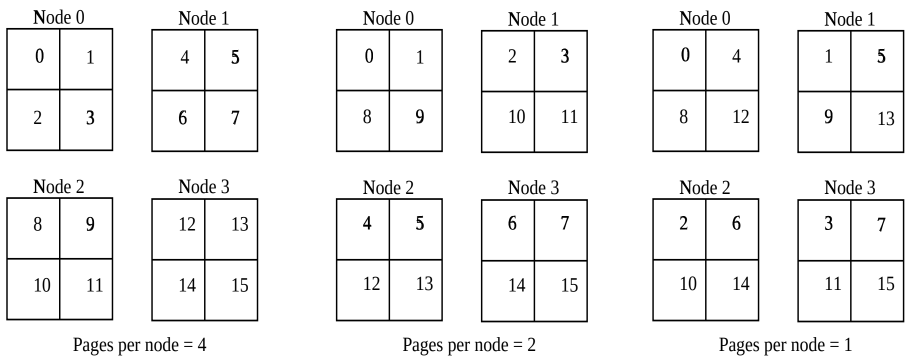

Figure 4: GTLB Mappings

4 Using the Hardware Mechanisms to Implement Shared Memory

Figure 5 shows the steps involved in completing a remote memory reference on the M-Machine when all of our hardware mechanisms are used. On cycle 1, a user program issues a load or store which references a remote address. By cycle 10, the event system has determined that the referenced block is remote and has started the event handler to resolve the event. By cycle 33, the event handler has decoded the type of the event and jumped to the correct routine to resolve it.

On cycle 49, the event handler completes the computation of the configuration space address which will be used to resolve the original load or store, and executes a **GPRB** operation to probe the GTLB for the home processor of the requested address.

In cycles 63-111, the event handler creates a record describing the remote operation and enqueues it in the software pending operation structure that records all remote memory references being completed in software. It then sends the request message to the home node and terminates. The request message must be sent after the pending operation data structure has been unlocked to avoid a potential deadlock with the reply handler.

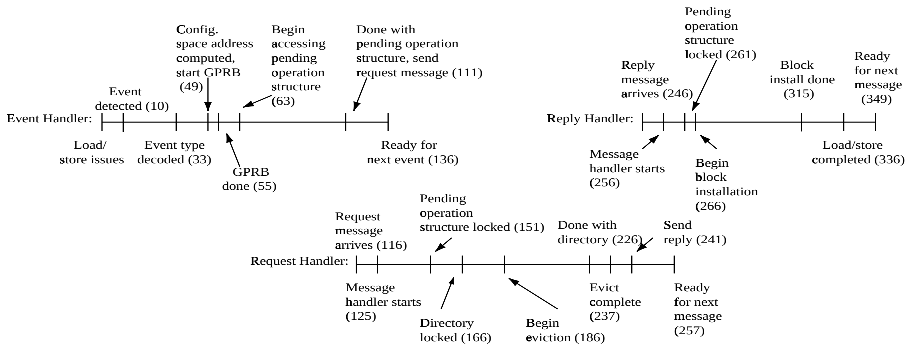
Event Handler:
Load/ store issues
Event detected (10)
Event type decoded (33)
Config. space address computed, start GPRB (49)
GPRB done (55)
Begin accessing pending operation structure (63)
Done with pending operation structure, send request message (111)
Ready for next event (136)
Reply Handler:
Reply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

Figure 5: Remote Request Timeline

The request message arrives at the home processor on cycle 116. The next 50 cycles are spent locking data structures, to ensure that no other thread modifies the state of the referenced block while the request handler is executing. On cycle 186, the eviction of the home processor's copy of the block begins, which completes on cycle 237. The reply message containing the requested block is sent on cycle 241.

On cycle 246, the reply message arrives at the requesting processor. Block installation begins on cycle 266. On cycle 315, the installation completes, and the reply handler begins to resolve the load or store that caused the remote reference, completing the original operation on cycle 336.

5 Evaluation

A three-hop cache-coherence protocol was implemented in software on the MSIM simulator to evaluate our mechanisms for shared memory. MSIM is a C-language model of the M-Machine that gives execution times within 10% of those given by the cycle-accurate RTL model of the MAP chip on our verification test suite. The model of the MAP chip used for these experiments has a 128-entry, two-way set-associative LTLB as well as floating-point units in all three clusters, restoring features that were removed late in the implementation process due to area constraints. The shared-memory handlers use the floating-point registers as temporary storage and perform constant generation in the floating-point units, making them relevant for this study.

The handlers used for these experiments were implemented in hand-coded assembly language. Four versions of the handlers were written, one which uses all of the hardware mechanisms, one which only uses the block status bits, one which uses the block status bits and the GTLB, and one which uses the block status bits and the dedicated thread slots for software handlers. Versions which did not use the GTLB included a software address translation routine, while versions which did not use the thread slots simulated context switches when starting or exiting handlers.

5.1 Remote Access Times

Figure 6 shows the M-Machine's remote access time as a function of the set of mechanisms used. The column labelled "Full M-Machine Mechanisms" shows the remote access time when all of the hardware mechanisms are in use, measured from the cycle on which the processor issues a load to the cycle on which an instruction which uses the result of the load issues. Proceeding to the left, the columns show the remote access time if various subsets of the mechanisms for shared-memory are used. The leftmost column shows an estimate of the remote access time if none of the MAP chip's mechanisms are used, while the rightmost column shows the remote access time of an 8-processor SGI Origin 2000 [11] for comparison purposes. All of the columns which show measured data from the M-Machine have been subdivided into the time spent in the event handler on the requesting processor, the request handler on the home processor, and the reply handler on the requesting processor.

Based on block transfer programs written for the M-Machine, we estimate that using the block status bits reduces remote access time by approximately 1000 cycles when only one block of a page is required. This estimate was used to generate the leftmost column of Figure 7. If a program requires fine-grained sharing of data, this latency reduction can significantly improve performance. For programs with more coarse-grained data sharing, using software to implement shared memory allows the block size of the protocol to be increased to match the needs of the application.

If the GTLB is added to the block status bits, remote access time is reduced to 427 cycles, an improvement of 18%. Using the block status bits and the dedicated thread slots for software handlers has similar results, reducing the remote access time to 433 cycles (17% improvement). Combining the block status bits, the GTLB, and the dedicated thread slots for shared-memory handlers so that all of the M-Machine's hardware

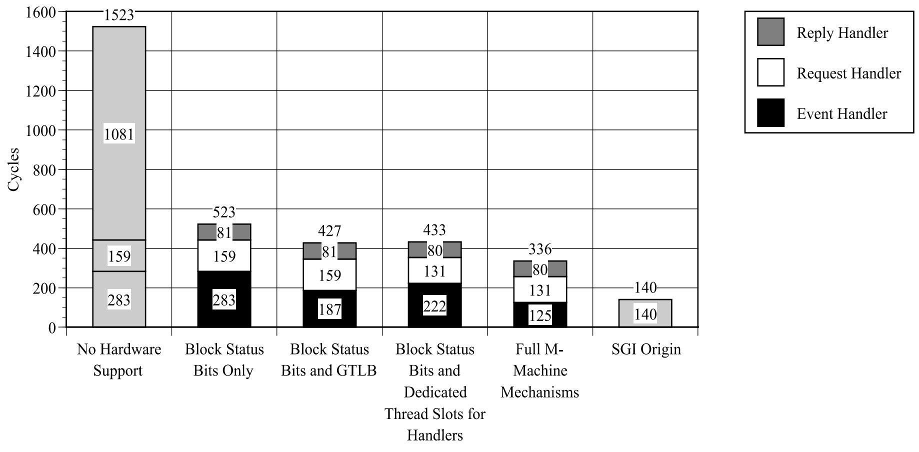

Figure 6: Remote Access Times

mechanisms are in use gives a remote access time of 336 cycles, a 35% improvement over the block status bits alone.

The mechanisms affect the execution time of different handlers in the shared-memory protocol, which has a significant impact on program performance, as will be shown later. When the GTLB is used, only the execution time of the request handler changes, since the determination of the requested address' home processor is only done once, in the request handler. On the other hand, using the dedicated thread slots affects the execution time of both the event and the request handlers, as it eliminates context switch overhead from all of the handlers. The execution times for the reply handlers shown on this graph do not change when the dedicated thread slots are used because execution time is measured from the point at which the first instruction of a handler executes and the context switch at the end of the reply handler does not affect the total latency.

Figure 7 shows how the remote access time changes when one or more invalidations are required to complete a remote reference. In the protocol implemented for these experiments, the home processor does not send an invalidation message to itself when it needs to invalidate its copy of a block, so the points shown on the graph represent 2-, 4-, and 8-way sharing of data. All versions of the protocol see an increase in remote access time of approximately 50% when an invalidation is required to complete a remote memory reference. As the number of invalidations required to complete the reference increases, all of the protocols see a linear increase in remote memory latency, because the bottleneck is the time required to process the acknowledgement message from each invalidation on the requesting processor.

Use of the GTLB produces a constant improvement in remote access time, independent of the number of invalidations, while the dedicated thread slots give an improvement which increases with the number of invalidations. Again, this is due to the fact that the determination of the home processor of an address is only done once. In contrast, the use of dedicated thread slots eliminates the context switch overhead from

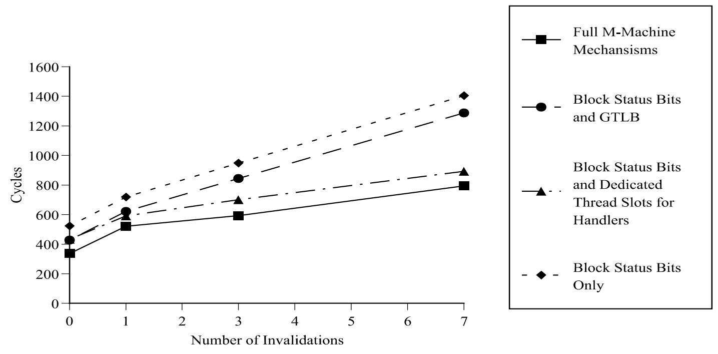

Figure 7: Remote Access Time With Invalidations

each handler. As the number of invalidations increases, the number of handlers in the critical path of the remote request increases, and therefore the performance impact of the dedicated thread slots increases with the number of invalidations.

5.2 Program Results

Two programs were simulated to evaluate the impact of the mechanisms for shared memory on program execution time: a 1,024-point FFT, and an 8x8x8 multigrid computation. These programs were based on source code provided by the Alewife group at M.I.T. [2], and were written in C with annotations for parallelism. The measurements presented here show the execution time of the parallel kernel of each application.

These programs display very different shared-memory characteristics. In FFT, remote memory references are relatively infrequent and are fairly evenly distributed across the processors. Multigrid makes many more remote references, and spends much more time waiting for memory than FFT. More importantly, multigrid's remote memory references are poorly distributed across the processors, because the destination matrix fits in a single page of memory and is thus mapped onto only one processor. Due to this difference in memory access patterns, the shared-memory performance of FFT is dominated by the latency of the shared-memory handlers, while the performance of multigrid is dominated by the occupancy of the request (priority 0 message) handler slot on the hot-spot processor.

Figure 8 shows the execution time of a 1,024-point FFT on the M-Machine. As would be expected from the low demands that this program places on the shared-memory system, overall performance is good, achieving better than 4 times speedup on eight processors even when only the block status bits are used. Adding either the GTLB or the dedicated thread slots for software handlers to the block status bits improves performance by 2-5%, depending on the number of processors. Adding both of these mechanisms gives speedups almost equal to the speedups provided by each of the mechanisms independently, reducing execution time by up to 9%. .

Figure 9 shows the execution time of an 8x8x8 multigrid computation. When just the block status bits and the GTLB are used, execution time is up to 20% greater than

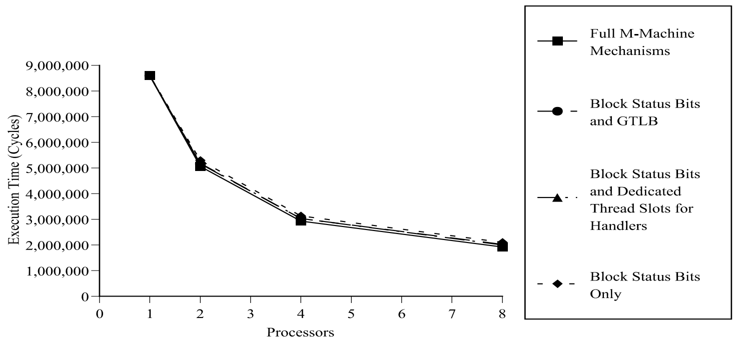

Figure 8: 1,024-Point FFT

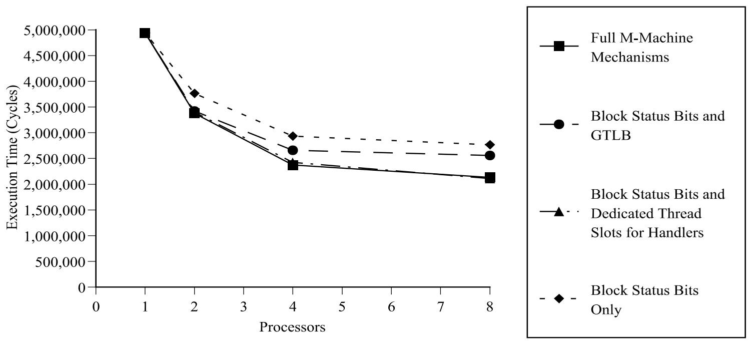

Figure 9: 8x8x8 Multigrid

when all of the M-Machine's mechanisms are in use. Using only the block status bits increases execution time by up to 30% over the full-mechanism case.

Interestingly, disabling the GTLB so that only the block status bits and dedicated thread slots are used does not significantly increase the execution time of multigrid. In fact, performing address translation in software gives a 1% performance improvement over using all of the hardware mechanisms when the program is run on eight processors. This counter-intuitive behaviour occurs because the GTLB reduces the execution time of the event handler on the requesting processor, while the dominant factor in Multigrid's performance is the occupancy of the request handler thread on the hot-spot processor, which is not affected by the use of the GTLB. In fact, using the GTLB increases the rate at which requests arrive at the hot-spot processor, increasing the number of requests which must be returned to the requesting processor because the block they request is in the process of being invalidated. These requests must be retried later, increasing the occupancy of the request handler on the hot-spot processor and decreasing overall performance.

6 Related Work

A number of systems have explored different hardware/software tradeoffs in implementing distributed shared memory. IVY [13] implements shared memory in software through user-level extensions to the virtual memory system. Shasta [16] and Blizzard-S [17] rely on the compiler to insert check code before each memory reference to determine whether the referenced data is local or remote. Blizzard-E [17] takes a different approach, modifying the error correction code (ECC) bits on blocks of data to force traps to software when remote blocks are referenced.

These systems show the advantages of the MAP chip's base architecture over commodity processors in implementing software shared memory. Blizzard-S and Blizzard-E have remote memory latencies of approximately 6000 cycles, while Shasta achieves latencies as low as 4200 cycles. In contrast, the M-Machine's remote memory latency is estimated to be approximately 1500 cycles when the event system is the only mechanism in use, giving it a 2.8-4x speed advantage over these software-only systems. This speed advantage is due to the MAP chip's event system and integrated network hardware, which reduce the time to invoke software handlers and inter-processor communication delay. Adding the block status bits, GTLB, and dedicated thread slots for software handlers to the base MAP architecture reduces the remote access time to 336 cycles, increasing the M-Machine's advantage to 12.5x-17.8x.

The Typhoon [14][15] and FLASH [10][6][7] projects explored the use of a dedicated co-processor to implement shared memory. Typhoon used a commodity processor to execute the shared-memory handlers, while FLASH relied on a custom MAGIC chip. Typhoon-0, the least-integrated of the systems studied in the Typhoon project, was implemented using FPGA technology, and achieved a remote memory latency of 1461 cycles. Two more-integrated versions of the Typhoon architecture, Typhoon-1 and Typhoon, were studied in simulation, and had remote memory latencies of 807 and 401 cycles respectively. With all of its mechanisms in use, the M-Machine has better than a 4x advantage in remote memory latency over Typhoon-0, and a 19% advantage over the full Typhoon system. Most of this advantage comes from the M-Machine's superior network subsystem and low-latency event system.

FLASH is able to complete a remote memory access in 111-145 cycles, depending on whether the remote data is cached. The ISA of the MAGIC chip is a major contributor to FLASH's low memory latency, as it contains many non-standard instructions to accelerate shared-memory protocols. In [6], the authors report that at least 38% of protocol processor issue slots contain one of these non-standard instructions, suggesting that the latency of shared-memory handlers running on MAGIC is significantly reduced by the addition of these instructions.

Comparing the M-Machine to these other systems shows the advantages of integrating hardware support for software shared memory into the processor. The M-Machine achieves significantly better remote memory latencies than the software-only shared memory systems by performing common tasks in hardware. In addition, the M-Machine has better remote memory latencies than any of the Typhoon systems, in spite of the fact that they utilize substantial custom hardware and a commodity co-processor to implement shared memory. While FLASH's remote memory latencies are significantly better than the M-Machine's, much of this is due to the MAGIC chip's optimized instruction set, creating an opportunity for future work which combines hardware support for software shared memory with an optimized instruction set.

7 Conclusion

In this paper, we have shown that adding a small set of hardware mechanisms to support software shared memory to a processor can significantly improve remote memory latency at low hardware cost. We have implemented four key mechanisms for software shared memory: block status bits to allow small blocks of data to be transferred between processors, a fast event system to detect remote memory accesses and invoke software handlers, dedicated thread slots to eliminate context switch overhead when starting handlers, and a global translation lookaside buffer to determine the home processors of remote addresses in hardware. In combination, these mechanisms allow remote memory accesses to be performed in as little as 336 cycles, significantly faster than most software-only or combined hardware/software shared memory systems. Hardware cost of these mechanisms is small -- approximately 3.5KB of storage, and some control logic.

Program-level experiments showed that the impact of our mechanisms on program execution time depends strongly on whether the dominant factor in the program's performance was the latency or the occupancy of the shared-memory handlers. On a 1,024-point FFT, in which latency was the dominant factor, using all of the mechanisms improved performance by up to 9% when compared to using only the block status bits. On an 8x8x8 multigrid, which is dominated by the occupancy rate of the request handler on the hot-spot processor, the mechanisms improved execution time by up to 30%.

The mechanisms implemented on the MAP chip substantially improve the M-Machine's shared-memory performance at a low hardware cost. However, the M-Machine's remote access time is still more than a factor of 2.5x greater than that of contemporary full-hardware shared-memory systems, suggesting that additional hardware support is required to close the performance gap between hardware- and software-based shared-memory systems.

8 References

[1] Anant Agarwal *et al.* The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[2] Ricardo Bianchini. Application Performance on the Alewife Multiprocessor. Alewife Systems Memo #43, LCS, Massachusetts Institute of Technology, 1994.

[3] 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,* October 1994.

[4] Nicholas P. Carter. Processor Mechanisms for Software Shared Memory. Ph.D. Thesis, Massachusetts Institute of Technology, 1999.

[5] 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,* Ann Arbor, MI, December 1995.

[6] 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), October 1994

[7] 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*, October 1994

[8] 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*, May 1992.

[9] 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.

[10] Jeffery Kuskin *et al.* The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual Symposium on Computer Architecture,* June 1994

[11] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual Symposium on Computer Architecture*, June 1997.

[12] 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*, November 1998.

[13] Kai Li. IVY: a Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, 1988.

[14] 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*, April 1994.

[15] 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*, May 1996

[16] 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*, October 1996.

[17] 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*, October 1994.