TMA: A Trap-Based Memory Architecture

Håkan Zeffer, Zoran Radović, Martin Karlsson and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, SWEDEN
{hakan.zeffer, zoran.radovic, martin.karlsson, erik.hagersten} @it.uu.se

ABSTRACT

The advances in semiconductor technology have set the shared-memory server trend towards processors with multiple cores per die and multiple threads per core. We believe that this technology shift forces a reevaluation of how to interconnect multiple such chips to form larger systems.

This paper argues that by adding support for *coherence traps* in future chip multiprocessors, large-scale server systems can be formed at a much lower cost. This is due to shorter design time, verification and time to market when compared to its traditional all-hardware counter part. In the proposed *trap-based memory architecture* (TMA), software trap handlers are responsible for obtaining read/write permission, whereas the coherence trap hardware is responsible for the actual permission check.

In this paper we evaluate a TMA implementation (called *TMA Lite*) with a minimal amount of hardware extensions, all contained within the processor. The proposed mechanisms for coherence trap processing should not affect the critical path and have a negligible cost in terms of area and power for most processor designs.

Our evaluation is based on detailed full system simulation using out-of-order processors with one or two dual-threaded cores per die as processing nodes. The results show that a TMA based distributed shared memory system can perform on par with a highly optimized hardware based design.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: PERFORMANCE OF SYSTEMS—*Design Studies*; C.5.5 [**Computer Systems Organization**]: COMPUTER SYSTEM IMPLEMENTATION—*Computer System Implementation, Servers*; C.5.1 [**Computer Systems Organization**]: COMPUTER SYSTEM IMPLEMENTATION—*Large and Medium Computers, Super Computers*; C.1.4 [**Computer Systems Organization**]: PROCESSOR ARCHITECTURES—*Parallel Architectures, Distributed Architectures.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS06, June 28-30, Cairns, Queensland, Australia.
Copyright C 2006 ACM 1-59593-282-8/06/0006 ...$5.00.

General Terms

Design, Performance.

Keywords

Server Design, Node Coherence Checks, Low Complexity Server Design, Software Coherence, Distributed Shared Memory (DSM), Trap-Based Memory Architecture (TMA), Chip Multi Processor (CMP), Simultaneous Multi-threading (SMT).

1. INTRODUCTION

While large-scale hardware-based shared memory systems have been successfully built for many years, the cost in terms of design and verification for each new generation is ever increasing. This is, for example, due to multiple levels of coherence, i.e., intra-snoop and inter-snoop domain coherence. Recently, the continued decrease in transistor size and the increasing delay of wires have lead to the development of chip-multiprocessors (CMPs) [3, 22, 31]. With the introduction of Sun's 32-threaded Niagara chip [14] what we used to know as mid-range servers actually do fit on a single die. This raises a very interesting question: what is the most cost-efficient way to connect multiple such chips together to form larger shared memory systems without sacrificing performance?

The traditional way of designing a large-scale shared-memory system is by modifying the memory system, leaving the processor core unchanged. We propose the opposite; adding a minimal amount of hardware support, contained within each processor core, combined with an entirely unmodified memory system designed and optimized for single-chip performance. Our proposal is an extension of existing trap mechanisms that enable efficient software handling of coherence. We believe that the cost of these modifications and additions in terms of area, power and engineer-years, are small enough to justify incorporating them in designs spanning multiple market segments. The goal being a system design cost and a system time-to-market that is equal to the processor design time. We call our proposal a "trap-based memory architecture," or simply TMA. This system architecture presents a completely binary-transparent view to the application and all necessary software support is contained in system software. While a software coherence scheme introduce some extra overhead when compared to a hardwired system, it also comes with some very attractive properties. The hard limit on the number of nodes in the system is removed, protocol bugs can be fixed with software patches and it is possible to change coherence schemes in a trivial manner.

Since area and complexity are scarce resources in a processor design, this paper evaluates a minimalistic TMA implementation based on simple coherence trap support and an incoherent commodity interconnect (e.g., InfiniBand [11]). Our proof-of-concept implementation, called TMA Lite, is evaluated with detailed full-system simulation of systems formed from chip multiprocessors, where each core is a superscalar, multi-threaded, dynamically scheduled out-of-order processor. We find TMA performance to be competitive with a highly optimized hardware-based distributed shared memory (DSM) system, with a vastly higher degree of complexity and a significantly longer design time.

Even though trap-based coherence can be used across the entire application domain, we believe that it is especially well suited for system designs targeting the high-performance computing market. This is partly because the recent trend towards cluster-based systems where TMA can provide shared memory and partly because the data access pattern regularity, often displayed by scientific codes, can be exploited by optimized coherence schemes.

The contribution of this paper is fourfold:

- We present an application binary transparent system based on fine-grained software coherence.
- We introduce a new architecture, the trap-based memory architecture (TMA) and a low-complexity implementation (TMA Lite) where all modifications are kept within the processor.
- We propose a new processor hardware structure to accelerate write permission checks, called write permission cache (WPC), and a comparator-based hardware structure for detecting read permission violations.
- We present a full-system simulation evaluation using out-of-order SMT chip-multiprocessors, showing that a TMA system with optimized coherence routines performs on par with a hardware system.

The core of the trap-based memory architecture is to detect fine-grained coherence violations in hardware, trigger a coherence trap when one occur and maintain coherence by software in coherence trap handlers. The detection and handling of coherence violations can be implemented with a minimal amount of extra hardware support by exploiting the trap mechanisms already existing in modern microprocessors.

A system based on coherence traps can be implemented at a fraction of the cost of the corresponding large-scale hardware DSM system, while offering similar performance. Our approach also removes the limit on the number of nodes, since the coherence protocol is implemented in software and all directory state resides in main memory.

2. PROCESSOR SUPPORT

This section describes load and store trap support and discusses some implementation issues. A coherence trap is in many ways similar to a page miss in a software managed TLB. When a coherence violation is detected, a trap is signaled and the effective address of the faulting load or store is forwarded to the trap handler. The coherence protocol code in the coherence trap handlers are based on the DSZOOM system [24] and are further described in Section 3.2. The coherence trap handler then fetches valid data and obtains read/write permission. After completing the coherence action, the faulting instruction is retried.

2.1 Load Support: “Magic” Value Sentinel

In the TMA Lite system, all coherence units in state invalid store a predefined “magic-value” as independently proposed by Scales et al. [26] and Chiou et al. [8]. Read misses can therefore be detected by comparing the value returned by each load to this predefined value and trigger a read miss coherence trap whenever a match is found. Note that a load coherence trap does not always indicate a coherence miss. This might happen if the loaded value is (and should be) the magic value. These situations have been shown to be rare and are simple to detect and handle within the protocol code [26].

2.2 Store Support

The simplest form of store permission support would be to trap on all stores. This strategy has the drawback of sometimes taking traps when the node already has store permission. Ideally, a coherence trap should only occur when the node does not have store permission. In this study, we model these two “extremes” and a hardware store support mechanism based on the previously proposed software write permission cache [35]. If an address tag is present in the WPC, it implies that the node already has write permission for the coherence unit it represents. Hence when using a WPC, the check for store permission must only be made when a store misses in the WPC.

The hardware WPC offers binary transparency to the system. It can be seen as a small (e.g., 8 or 16 entries) virtually indexed, virtually tagged cache, that is looked up for each store. When the effective address of a store is not present in the WPC, the store is marked as faulting. The faulting instruction will then invoke the write miss trap handler, which will obtain permission and fill the WPC.

2.3 Implementation Issues

While many optimizations can be implemented to speed up the coherence trap handling, this paper focuses on an implementation with the bare minimum set of hardware changes. The TMA Lite implementation relies on the existing exception trap vector mechanism as defined by SPARC-V9 [33] and use reserved entries in the trap vector for the coherence trap protocol code. Register spill is avoided by using the *alternate global* registers [33]. That is, no state has to be saved as long as no context switch occurs. The WPC structure can be manipulated through ASI-mapped registers, requiring only a new ASI from an ISA perspective.[1]

We believe that the load coherence trap can be implemented in many modern processor designs with negligible cost in area and complexity. However, the target processor must be able to take a trap when the read value is returned. That is, a load instruction is not allowed to be committed/removed from the reorder buffer (ROB) until the value is returned. Another alternative could be a checkpoint-based design [13]. Similar techniques have also been described by Qiu et al. [23] and Cain et al. [5]. Because a load trap only

[1]Address Space Identifiers [33]

needs to be detected before the load is committed, not before the data is forwarded to other dependent instructions, it should be possible to implement without affecting the critical path.

The WPC can be implemented as a small cache accessed in parallel with the TLB[2] or as part of the TLB (similar to the RS/6000 TLB design [21]). We model the former and believe that the WPC-based store trap can be implemented in many modern processor designs with negligible cost in area and complexity. Since the WPC is accessed in parallel with the TLB, the WPC access should be off the critical path and not affect the cycle time nor the single thread performance (when turned off). Since our WPC implementation is virtually indexed and virtually tagged, the TMA Lite system can only provide extra scalability on the application level. An implementation using physical addresses, on the other hand, could provide real system level scalability and allow operating systems to boot across multiple nodes.

3. PERFORMANCE EVALUATION

This section describes our simulation methodology, simulator framework, benchmarks, results and the architectures modeled.

3.1 Simulation Methodology

We use the Simics full system simulator [19] extended with an out-of-order processor model, memory hierarchy and a system interconnect model [32]. We simulate a SPARC-V9 system running an unmodified Solaris 9 operating system. Our processor model is based on the Simics Micro Architectural Interface which guarantees correctness leaving timing modeling to processor and memory system models.

Core and Chip Model: We model a dual-core CMP where each core is a superscalar, dynamically scheduled, 2-way multi-threaded out-of-order processor. The processor model is implemented in the following way. Both threads fetch 8 instructions from a dual-ported instruction cache. The scan stage scan instructions in the fetch buffer until a predicted instruction falls outside the fetch buffer. As long as the program execution follows the fall through path, no fetch bubbles are generated. When a branch is taken, a one cycle fetch bubble is generated before the scan unit is able to redirect the instruction fetcher. The two threads share branch-target buffer (BTB) and branch predictor, but have separate return address stacks (RAS). Instructions are selected from the dual fetch unit based on the ICOUNT fetch policy and inserted into the 6-way superscalar decode-rename stages. After register renaming, instructions are put in the issue queue. Instructions may be issued and executed out of order, but are committed in order. A round robin policy is used between the threads to select which instructions to commit.

Our memory hierarchy simulator models the latency and bandwidth of three levels of lockup-free caches per chip, where the second and third level is shared among cores. We model first-level write-through caches, while the second and the third level caches both implement a write-back strategy. The L3 latencies were chosen to model on-chip tags and off-chip data. Table 1 shows simulated chip parameters, which

Processor	2-way SMT, single- or dual-core
Frequency	3 GHz
Pipeline Stages	12
Fetch/Issue/Retire Width	16/6/8
Instruction Window	256
Store Buffer	32 entries per thread
L1 Data Cache	32KB 2-way, 4 MSHRs, 2 cycle hit
L1 Instruction Cache	64KB 2-way, 4 MSHRs, 2 cycle hit
L2 Shared Unified Cache	1MB 16-way, 16 MSHRs, 11 cycle hit
L3 Shared Unified Cache	8MB 16-way, 16 MSHRs, 81 cycle hit
L1/L2/L3 Block Size	64 bytes
Memory Latency	200 cycles, load-use
TMA Lite Store Support	16-entry hardware WPC
TMA Lite Load Support	Magic-value comparator

Table 1: Simulated chip parameters.

4-node Configuration	Dual-core chip, 4 threads per node
8-node Configuration	Single-core chip, 2 threads per node
Interconnect Bandwidth	3 GB/sec per link
Network Topology	Fully connected
Remote Memory latency	600 cycles, load-use

Table 2: Simulated system parameters.

were chosen to resemble a scaled down Power5 design.

System Configuration: Table 2 shows simulated system parameters. We evaluate our system using both a 4-node and an 8-node configuration, where each node consists of a single chip. The 4-node system is built from dual-core chips and the 8-node system from single-core chips. However, all cores contain 2 hardware threads regardless of system configuration. Hence, there are a total of 16 threads per system for both configurations.

Our nodes are fully connected to each others with a high-bandwidth, low-latency interconnect. Our network supports two different modes: a hardware-coherent and a non-coherent mode. The hardware-coherent mode supports the coherence messages used by the hardware-only system. While the non-coherent mode support mechanisms for *put*, *get*, and *atomic* operations to remote nodes' memories, similar to InfiniBand [11] or Sun Fire Link [29]. Remote operations are accomplished without interrupting remote processors.

Interactions with Solaris: To make the trap handling as realistic as possible, we use reserved trap types in the SPARC-V9 instruction set to implement our coherence traps. We have applied a binary patch that modifies the corresponding trap vector entry in Solaris 9 with our coherence protocol code.

The simulated load sentinel comparator and all three store-permission checks modeled can signal load and store instructions as faulting. A coherence trap is taken when the instruction marked as faulting reaches the commit stage. When a coherence trap occur, it is handled just as a normal trap and the protocol routines are executed just as any other trap handler instructions in the pipeline of the cycle-accurate simulator, consuming pipeline resources and polluting the caches.

TMA Lite relies on the operating system (OS) for detecting when a private page moves to shared state (handled with

[2]In the case of both a TLB miss and a WPC miss, the TLB miss takes precedence. The WPC miss will be triggered when the instruction is retried.

the virtual memory subsystem). The OS is also responsible for local memory mappings, network interface mappings and replication of pages in shared state. The coherence trap handlers are used to maintain fine-grained coherence between the "private copies" of all shared pages. For this to work we assume private/shared functionality (such as a bit) per TLB entry informing the coherence check hardware to trap on a miss or not. This functionality is already existent in many TLB designs and the information just has to be propagated to the coherence check mechanism.

3.2 Simulated Protocols

This section describes the hardware-only and the software-based coherence protocol (executed by coherence trap handlers) modeled in this paper.

Hardware Protocol (HW): The hardware-only protocol is a highly optimized non-blocking MOSI directory protocol. The on-chip coherence agent, responsible for node-to-node coherence, has dedicated directory memory and uses a fully mapped bit vector to keep track of sharers [17].

Software Protocol (SW): The software-based protocol is a port of the well tested DSZOOM protocol [35, 24], a distributed synchronous directory coherence protocol [9, 24, 35]. Our software protocol assumes a high bandwidth, low latency cluster interconnect, supporting fast mechanisms for *put*, *get* and *atomic* operations to remote nodes' memories, such as InfiniBand [11] or Sun Fire Link [29]. The protocol further assumes that the write order between any two endpoints in the network is preserved. These network assumptions make it possible for a trapping processor to get read/write permission without remote interrupt- and/or poll-based asynchronous protocol processing [4, 24].

A processor that has detected the need for global coherence activity (by a coherence trap) can lock a remote directory entry and independently obtain read and write permission. The protocol allows several threads in the same node to perform protocol actions at the same time. The invalidation-based protocol (MSI) directory is located in the nodes' memories. Effective address bits of memory operations determine the location of a coherence unit's directory location, i.e., its "home node."

To reduce the number of accesses to remote directory entries caused by WPC fill traps, each node has one byte of local state (MTAG) per global coherence unit indicating if the coherence unit is locally writable. The directory is only consulted if the MTAG indicates that the node currently does not have write permission to the coherence unit. Since a home node can detect with local memory accesses if it has write permission (the directory is located in its local memory), it does not need any extra MTAG state.

3.2.1 2-hop Write Miss Example

Figure 1 illustrates the protocol activity caused by a 2-hop write miss to coherence unit A and a 3-hop read miss to coherence unit B. The 2-hop write miss coherence activity is started when a processor in node1 writes to coherence unit A. The transactions are marked `A1-A3`. State transitions for coherence unit A and B are shown below each node.

HW: The coherence agent starts the coherence activity when the store operation is at the head of the store buffer and the miss is detected. The coherence agent sends a *read-exclusive-request* (`A1` in Figure 1 (a)) to the home node (node0). The home node responds with permission and data (`A2`). Finally, node1 sends an acknowledgment (`A3`) to the home node (off the critical path).

SW: The coherence activity is started since the requesting processor in node1 does not have write permission, and hence, takes a coherence trap. The requesting processor acquires exclusive access to A's directory located in node0's memory (indicated by the remote atomic operation `A1` in Figure 1 (b)). When the directory is locked, it retrieves the data from the home node with a remote get operation (`A2`). To end the coherence activity, the requesting processor releases and updates the directory with a single remote put operation (`A3`), which is off the critical path.

3.2.2 3-hop Read Miss Example

Figure 1 also illustrates protocol activity caused by a 3-hop read miss to coherence unit B. The coherence activity starts when a processor in node3 reads coherence unit B. The transactions are marked `B1-B5`.

HW: The coherence agent in node3 detects the coherence miss and sends a *read-request* to the home node (`B1` in Figure 1 (a)). The home sends an *intervention* to node2 (`B2`) which has coherence unit `B` in state modified. Node2 responds with data and permission information to node3 (`B3`) and enters owner state. Node3 ends the coherence activity by sending an acknowledgment (off the critical path) to the home node.

SW: The coherence activity starts with a load coherence trap triggered by the magic-value sentinel. The requesting processor in node3 locks the directory entry and determines the identity of the node holding the data (`B1` in Figure 1 (b)). The data happens to reside in node2, in a modified state. A second remote atomic operation (`B2`) to node2's MTAG structure disables write permission on that node. The data is fetched with a remote get operation (`B3`). Node2's MTAG and the home's directory are then released and updated. These two put operations (`B4` and `B5`) are off the critical path.

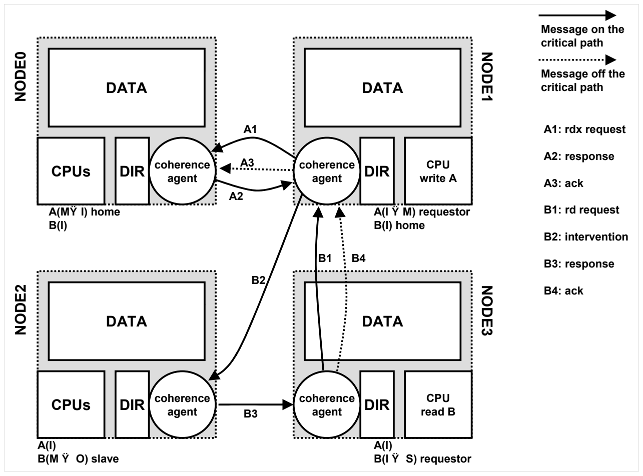
3.2.3 Hardware/Software Protocol Discussion

While handling the coherence protocol using trap mechanisms on the requesting processor has advantages, it also has some drawbacks. On most processors, handling a trap is associated with a significant pipeline disruption. For example, in our model, when a coherence trap is taken, all the trapping hardware thread's instructions are flushed from the pipeline. The trap handler has to be fetched and the instructions have to fill the pipeline frontend before the actual execution of the coherence routine is started. When the node has read/write permission, the faulting instructions is retried. That is, the instruction has to be fetched again and walk through the entire pipeline.

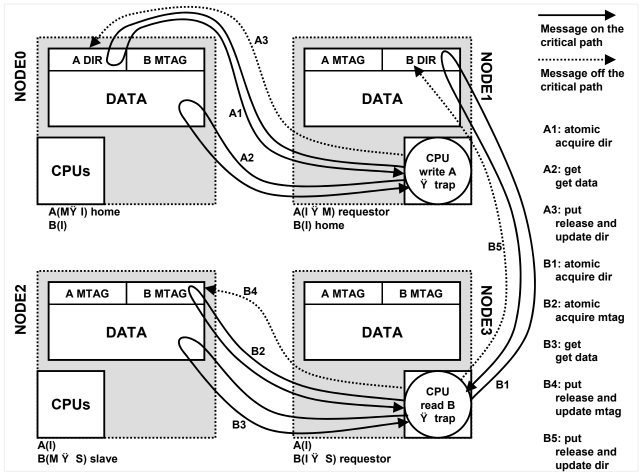
Our software protocol has two major drawbacks compared to its hardware-only counterpart. First, it needs more inter-node hops to solve coherence misses (see Figure 1). Second, the software store protocol is not capable of hiding store misses, like store buffers often can in traditional hardware protocols.

3.2.4 Memory Consistency and Deadlock Avoidance

Our coherence protocol maintains sequential consistency (SC) [16] by requiring all acknowledges from the sharing nodes to be received before a global store permission request is granted. The WPC does not weaken the memory



(a) **HW:** Transaction A1-A3 shows a 2-hop write miss. Transaction B1-B4 shows a 3-hop read miss.



(b) **SW:** Transaction A1-A3 shows a 2-hop write miss. Transaction B1-B5 shows a 3-hop read miss.

Figure 1: Protocol examples for a 2-hop write miss and a 3-hop read miss.

model, since the WPC protocol requires all the remotely shared copies to be destroyed before granting the write permission. It simply extends the duration of the permission tenure before the write permission is given up. Of course, if the memory model of each node is weaker than sequential consistency, it will dictate the memory model of the system. Our nodes, and hence the system, implements total store order (TSO) [33].

All WPC-related deadlock issues are solved by the coherence run-time system. A processor's WPC entries have to be released at synchronization points, context switches, at failures to acquire MTAG/directory entries and at thread termination. To avoid deadlocks caused by user-level flag synchronization, we also periodically flush all WPC entries. This flush can be accomplished by the operating system, for example, on timer interrupts or with a dedicated timer based trap mechanism. Throughout this study, WPC entries are flushed every 10,000 cycles.

3.3 Benchmarks

The benchmarks that are used in this paper are the well-known workloads from the SPLASH-2 benchmark suite [34]. Data set sizes for the applications studied can be found in Table 3. Since we evaluate our system on a cycle-accurate simulator, we had to restrict our evaluation to a subset of the SPLASH-2 benchmarks. The long turnaround time, up to a week of simulation per data point, is also the reason why we use small working set sizes.

It is not clear however which system is favored by small working sets. In the hardware-only system, the large third-level cache will capture most of the data set, and hence, almost no extra coherence traffic is generated because of limited sharing space. The TMA Lite system, on the other hand, might get a slightly increased WPC hit rate, but be penalized by the increased rate of synchronization events that forces WPC flushes in order to avoid deadlocks.

The selected programs were chosen to represent a variety of communication and synchronization requirements. For example, `fft` has a communication-intensive behavior and `radix` has a randomized store access pattern.

Program	Problem Size
`fft`	64k points
`lu-c`	512×512 matrices, 16×16 blocks
`lu-nc`	512×512 matrices, 16×16 blocks
`radix`	256k integers, radix 1024
`water-nsq`	512 molecules, 3 time steps
`water-s`	512 molecules, 3 time steps

Table 3: SPLASH-2 benchmarks.

We warm the caches similarly to Woo et al. [34]. All applications are compiled with a gcc-3.4.3 compiler (optimization level 3). PARMACS macros for locks and barriers are based on user-level *test&test&set* spin locks. Pause/Event macros are implemented with the POSIX Pthread library (only `radix` uses a small amount of pauses).

3.4 Simulation Results

We start our performance evaluation by comparing the performance of our coherence trap enabled TMA Lite system to the hardware DSM system. As shown in Figure 2, the performance of the TMA Lite system is on average 25 percent behind the hardware DSM. However, we will show in this section how to increase the performance of the TMA Lite system.

We have found the WPC hit rate to be critical for TMA Lite performance. We therefore establish a lower and upper bound (the two "extremes" from Section 2.2) on the performance that can be obtained with the type of store coherence mechanisms proposed in this paper. The lower bound is the performance obtained with an "optimal" WPC implementation, i.e., store traps are only generated if the node does not have write permission.[3] Conversely, we use an implementation trapping on all stores, i.e., no write permission

[3]Note that some unnecessary cycles are spent nevertheless filling WPC entries that never will be used.

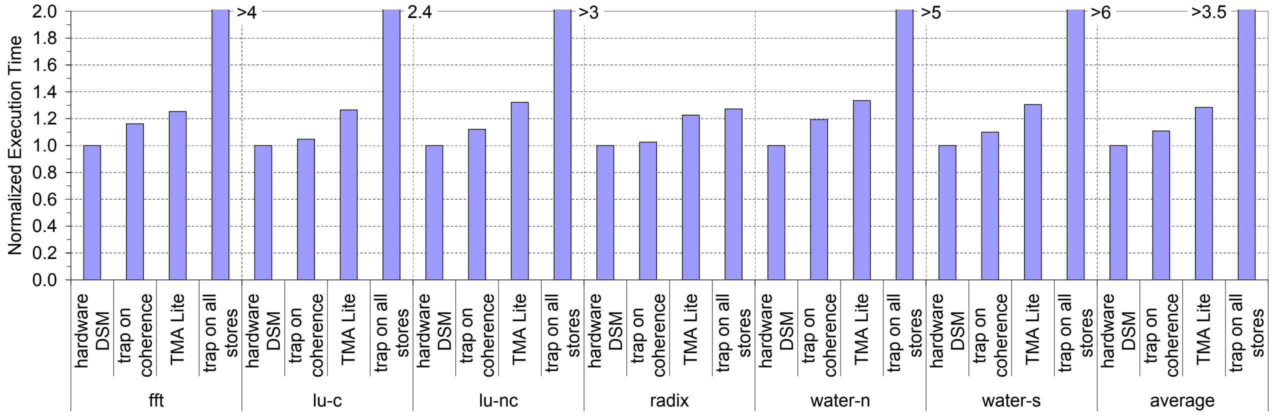


Figure 2: Software and hardware protocol performance. (4-nodes, coherence unit size: 64 bytes.)

caching is done at all, as an upper bound of the store coherence trap overhead. The lower and upper bound is shown in Figure 2 as `trap on coherence` (lower) and `trap on all stores` (upper). As can be seen in Figure 2, and as one might expect, the high penalty of taking a coherence trap makes trapping on all stores a costly strategy. Note that we have only trapped the processor on shared pages in this study and that the overhead would have been significantly higher without this capability. The performance of the `trap on coherence` bar, on the other hand, is very encouraging when compared to the hardware DSM system.

The hardware DSM system outperforms the lower-bound software for applications with a lot of coherence activity, this is for example the case for `fft`. On the other hand, when the amount of coherence activity is low, as in `water-s`, the optimal WPC performs on par with the hardware DSM system. The performance difference between the TMA Lite system and the optimal WPC comes from the fact that the WPC has to be filled, even though the node might already have permission. It has earlier been shown that the WPC hit rate for `radix` is very poor [35]. This is the reason why the WPC performs similar to the `trap on all stores` strategy for this particular application.

We find that we are able to get a significant performance effect from a simple 16-entry WPC compared to the trap-all strategy. The performance obtained by the optimal write permission caching makes more advanced WPC structures look very promising. This is however beyond the scope of this paper.

3.4.1 Trap Protocol Breakdown

In order to quantify and understand the different components of the software protocol, we have broken down the execution time into five different parts. Deadlock avoidance handling (described in Section 3.2.4), load protocol, store protocol, WPC fill and the remaining application execution denoted as other protocol cycles. We have divided the store handling into two different parts (*store protocol* and *wpc fill*) to highlight the impact of the overhead caused by WPC fills when the node already has permission. As can be seen in Figure 3 (a) the overhead caused by WPC fills is non negligible. For all benchmarks except `fft`, more time is spent filling the WPC than on the actual store protocol processing. The relative WPC cost is extra large in applications with little coherence activity.

We find that on average 31 percent of the overall execution time is spent on coherence actions. Out of which 7 percent is deadlock avoidance, 31 percent is load miss handling, 18 percent is store miss handling and 44 percent is WPC fill handling. To further understand the cost associated with coherence traps, we have quantified how much time is spent entering and exiting the trap handlers as well as the time spent waiting for remote get operations, lock handling and the overhead of remaining protocol handling.

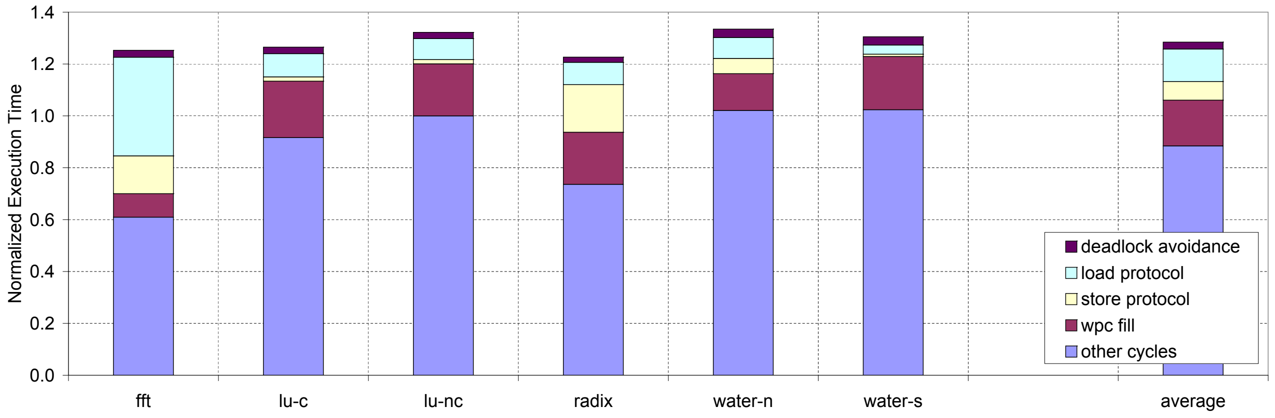
Figure 3 (b) shows a breakdown for each of the various forms of traps, again, we separate WPC fill processing from the store protocol. Each bar is divided into the following five parts: *trap enter*, *remote get*, *lock acquire*, *other protocol cycles* and *trap exit*.

Trap Enter: Represents the cost of filling the pipeline with the corresponding trap handler. It accounts for the total number of cycles from when the trap is taken until the first instruction in the trap handler is committed. A fetch unit optimization, similar to the one proposed in the SMTp proposal [7], can almost remove this part of the trap overhead.

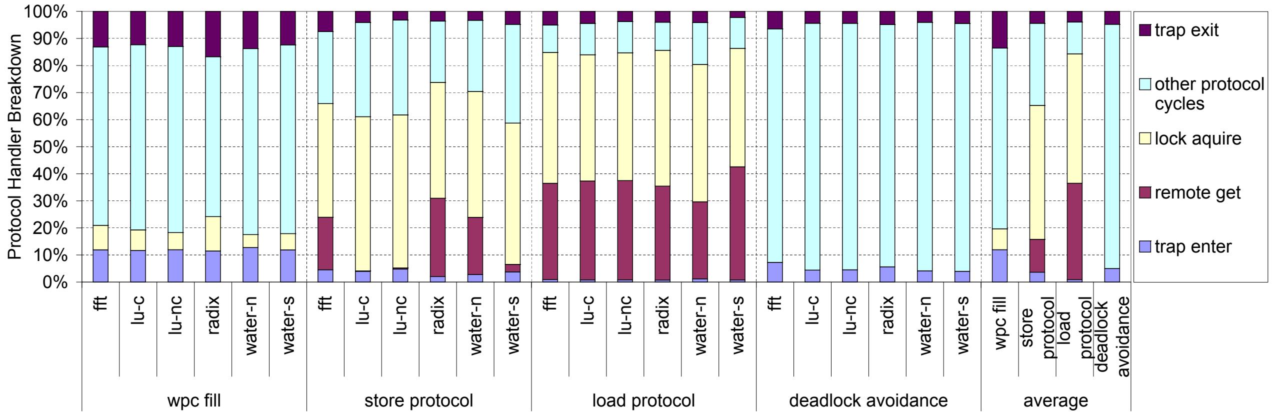
Remote Get: This part is the total number of cycles spent waiting for remote get operations to finish. That is, when the load or store protocol needs to get valid data. This part is, of course, highly dependent on the remote memory access time. A longer (shorter) remote memory latency will increase (decrease) this component of the trap handling.

Lock Acquire: Since the TMA Lite coherence protocol is based on software handlers, atomic updates of the directory state located in main memory are needed. The *lock acquire* part corresponds to the total number of cycles waiting for remote and local lock aquire operations while inside the protocol code. Note that the *lock acquire* part of the `wpc fill` bar corresponds to local lock operations to the MTAG structure.

Trap Exit: This part corresponds to the total number of cycles from the actual commit of the retry instruction ending the coherence trap routine until the pipeline is refilled and the instruction that originally caused the coherence trap to be taken is committed. A more advanced trap mechanism that would not require a pipeline flush when a coherence trap is taken, may decrease or completely eliminate this pipeline-refill time [12, 36].



(a) Execution time breakdown.



(b) Coherence protocol breakdown.

Figure 3: TMA Lite execution time and protocol breakdown. (4-nodes, coherence unit size: 64 bytes, 16-entry WPC, magic-value comparator.)

Other Protocol Cycles: The rest of the cycles of the coherence trap handlers are contained in the *other protocol cycles* part. These are typically cycles spent manipulating directory bits, setting up RDMA actions or updating the WPC. Note that the WPC is updated also in the `store protocol` bar.

Figure 3 (b) shows that the pipeline fill (*trap enter*) and the pipeline refill (*trap exit*) parts does not affect the performance that much. For a more disruptive trap mechanism this part may be significantly higher. Most of the cycles in the `wpc fill` bar are consumed updating the WPC (the *other protocol cycles* part). It is interesting to note that the overhead caused by local lock acquires to the MTAG structure is not severe, despite the costs associated with atomic operations. The `store protocol` bar, on the other hand, consumes a lot of its time waiting for lock acquires to finish. The reason why this takes so much time is because the directory often is located on remote nodes, and hence, the remote atomic fetch-and-set operation takes much more time than its local counter part. `fft`, `radix` and `water-n` all have a significant amount of write misses where a remote get of valid data is needed.

Most of the cycles in the `load protocol` breakdown are consumed waiting for exclusive access to the directory (*lock acquire*) and while getting valid data (*remote get*) from remote nodes. Because these two kinds of operations are very costly, the trap overhead (*trap enter* and *trap exit*) as well as bit manipulation overhead (the *other protocol cycles* part) are very small. Almost all cycles spent inside the deadlock avoidance handler are consumed while manipulating the WPC and releasing the locally cached locks.

3.4.2 *Coherence Unit Size Scaling*

Software-based coherence protocols have the advantage of making protocol changes trivial, enabling a degree of flexibility difficult to implement in hardware within complexity bounds. In this section we present the performance of the TMA Lite system while varying coherence unit size from 64-256 bytes.[4] We have broken down the execution time into protocol components, to expose how each component performs. As Figure 4 shows there are significant performance gains that can be achieved by tuning the coherence unit size. Except for `radix`, all applications benefit from some coherence unit scaling.

The greatest performance improvement is seen for `fft`

[4]The TMA Lite system allows the coherence unit size to be set at application launch.

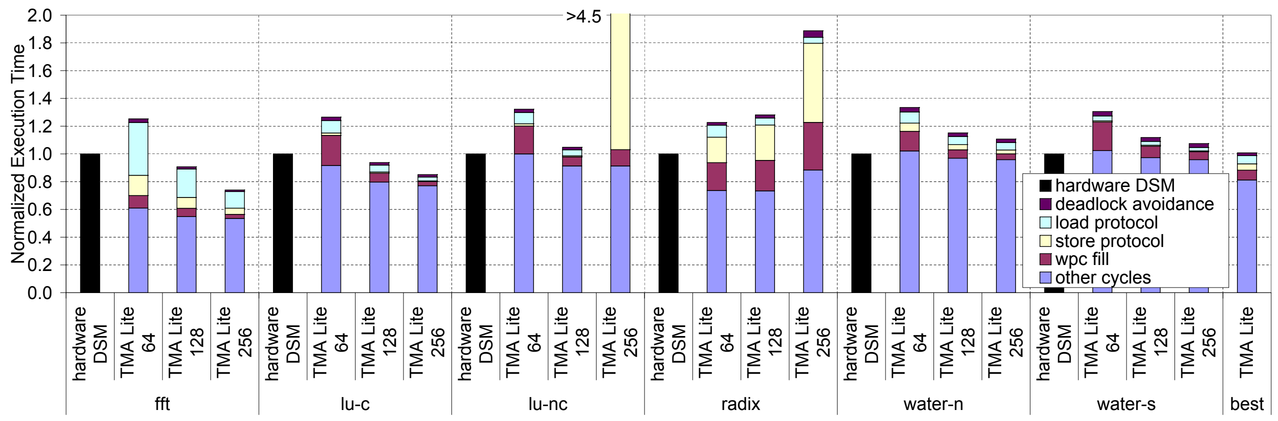


Figure 4: Normalized TMA Lite performance. (4-nodes, coherence unit size: 64-256 bytes, 16-entry WPC, magic-value comparator.)

and `lu-c`, whose load and store protocol cycles scale very well with coherence unit. The WPC fill cycles are also decreased, especially for `lu-c`. The performance improvement for the `water` applications comes from a 54 (60) percent WPC miss rate improvement (when going from 64-128 bytes) for `water-n` (`water-s`), and hence, a large reduction in WPC fill cycles. This is shown by the *wpc fill* part in Figure 4.

Both `fft` and `lu-c` outperform the hardware DSM system when a coherence unit size of 128 bytes is used. `fft` (`lu-c`) is more than 26 (15) percent faster than the hardware DSM system when a coherence unit size of 256 bytes is used. However, it is well known that not all applications scale with coherence unit size. `lu-nc`, for example, suffers from false sharing when a coherence unit size larger than 128 bytes is used. This is indicated by the *store protocol* component increase observable in Figure 4. The reason why the *other cycles* part is not constant between TMA Lite configurations while varying the coherence unit size is because extensive trapping affects both threads in the pipeline. More resources are freed/used, the fetch unit has to be redirected, etc.

The right-most bar in Figure 4 shows the average TMA Lite system performance when hand picking the best coherence unit for each application. The TMA Lite system performs within one percent of the hardware DSM. More advanced protocol optimizations will further improve TMA Lite performance.

3.4.3 Node Scaling

In order to test the scalability of the coherence trap proposal, we have modeled both 4- and 8-node hardware DSM and TMA Lite systems. We have chosen to use a total count of 16 hardware threads in all our configurations to avoid application scalability interference in our results. The 4-node configuration consists of four dual-core chips while the 8-node configuration consists of eight single-core chips. The 8-node system will has less node locality than its 4-node counter part, which will lead to more coherence activity.

By comparing the relative 4- and 8-node performance difference between the DSM and the TMA Lite system, we can get an indication of how the TMA Lite proposal scales compared to the DSM system. As can be seen in Figure 5, the TMA Lite system scales similarly to the hardware DSM system, except for `water-s`. For `water-s` we have noted an increase in load and store protocol cycles, however not enough to explain all of the performance degradation.

4. RELATED WORK

There is a wide range of options for hardware/software trade-offs for implementing coherent shared memory, ranging from all-hardware coherence to implementations relying on no specific hardware support at all. It is fairly common, however, that systems rely on both hardware as well as run-time software to improve the performance. Most cache-coherent non-uniform memory access machines (CC-NUMAs) rely on hardware performance counters to guide the page migration software [17, 9]. The Sun WildFire system has hardware support for detecting pages in need of replication [9]. Replicated copies of each page can be instantiated by software, but the coherence between the multiple copies is kept by hardware on a cache-line sized basis.

The hardware/software boundary is also sometimes crossed in order to handle some of the corner cases of the coherence protocol. One such example is the MIT Alewife machine that added efficient support for trapping to a software handler on the rare event of massive sharing [1]. These systems rely on hardware coherence and use software to simplify part of the design. Our proposal removes all of the inter-node hardware protocol from the memory system and rely entirely on coherence traps and software handlers for inter-chip coherence.

The trade-off between hardware/software is even more apparent in systems with programmable coherence engines or dedicated coherence processors such as Stanford's FLASH [15], Sun's S3.mp [20] and Wisconsin's Typhoon-0 [25]. Here, the entire coherence protocol is controlled by specialized software, which enable flexible protocol adoptions as well as protocol bug correction. SMTp is a more recent proposal [7] in which the coherence protocol is run by one SMT thread that handles coherence actions on processor cache misses. While these systems rely on software handlers for coherence, they require either dedicated coherence processors capable of snooping the memory bus [15, 20, 25] or extended memory controllers together with SMT processor pipeline modifications [7] in order to work. The TMA Lite

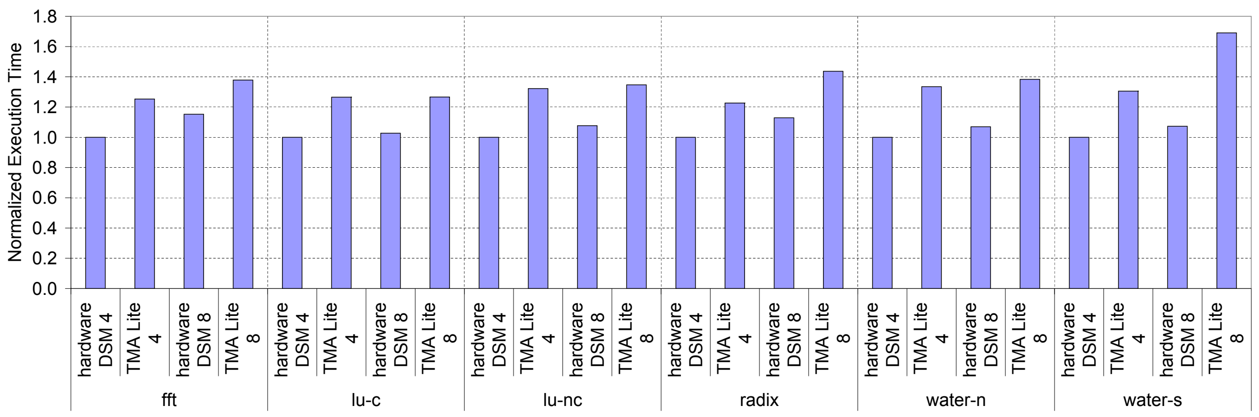


Figure 5: Normalized TMA Lite performance. (4/8-nodes, coherence unit size: 64 bytes, 16-entry WPC, magic-value comparator.)

proposal on the other hand leaves the memory system completely unmodified while adding two very small and simple hardware structures to the processor.

Page-based software-only DSM systems [18, 6, 2, 30, 4] rely on the existing TLB hardware to detect permission violations. However, TMA is targeting a much more fine-grained coherence granularity than a virtual memory page. An appealing option in a page-based software DSM is to extend the functionality of the TLB to support fine-grain load and store access control bits. For example, the RS/6000 [21] provides a single access control bit for every 128-byte segment. This would allow the fast trap mechanisms that are already present for handling TLB faults to also handle the coherence faults. This scheme has the advantage of an earlier detection than most other schemes, including our read support. However, such modifications of the TLB are less practical for TLBs with large or mixed page sizes.

The Blizzard-E [27] system demonstrated the use of the existing trap mechanisms to handle read coherence violations. By corrupting the ECC bits of coherence units in state invalid, an ECC trap is generated for all read misses. Then once the ECC trap is triggered the trap handler obtains read permission and corrects the ECC code. While having the advantage that no “false” read coherence traps are generated, the scheme have the drawback of requiring a network interface capable of setting this special ECC value in remote nodes[5] if a synchronous coherence protocol is to be used. While this approach may be suitable for read checks, it is less practical for writes. In the Blizzard-E system as well as in other fine-grained systems such as Shasta [26], Sirocco-S [28] and DSZOOM [24], permission checks were instrumented in the application code thereby breaking the application binary transparency targeted by the trap-based memory architecture.

The “Informing” memory operation proposal [10] is closest to our goal: to keep added hardware complexity low. While this is a very interesting approach for software coherence, the out-of-order processor implementation is non trivial. The coherence protocol assumes permission for all data in the L1 cache. However, an informing load operation can update the cache and later be squashed from the pipe (due to control speculation or a preceding exception) leaving the cache with data that have not been checked for permission. Horowitz et al. permits the speculative informing load operation to update the cache on a miss and makes sure that this data can not be read by another informing load operation by extending the data forwarding mechanism [10]. Shared L1 caches is likely to further complicate this scheme.

5. CONCLUSIONS

This paper explores a new design point in the trade-off between between hardware and software to implement shared memory. We call our proposed architecture, a trap-based memory architecture (TMA). In TMA, coherence trap hardware is responsible for performing the permission check and to generate a trap whenever a coherence violation is detected. Coherence is then maintained in software by coherence trap handlers, allowing a greater protocol flexibility than hardware implementations.

In this paper we evaluate a TMA implementation called TMA Lite. TMA Lite is targeted at minimizing system complexity, we have therefore assumed a commodity interconnect and employed a synchronous protocol. While the traditional way of designing a DSM system is by modifying the memory system leaving the processor unchanged, in TMA Lite we propose the opposite; adding support for *coherence traps* inside the processor core while leaving the memory system unchanged. We believe the proposed processor extensions; a hardware write permission cache (WPC) and a magic value check for loads, can be added to many processor designs with a very limited cost in terms of area, power and complexity.

Detailed full-system simulation of 4- and 8-node systems, based on chip multiprocessors, shows that our TMA Lite system can perform on par with a highly optimized hardware DSM system running on the same node and interconnect hardware.

[5]A capability currently not supported in e.g. the InfiniBand standard.

6. REFERENCES

[1] A. Agarwal et al. The MIT Alewife Machine. *IEEE Proceedings*, 1999.

[2] C. Amza et al. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] L. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *ISCA'00*, pages 282–293, June 2000.

[4] A. Bilas, C. Liao, and J. P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *ISCA'99*, pages 282–293, May 1999.

[5] H. W. Cain and M. H. Lipasti. Memory Ordering: A Value-Based Approach. In *ISCA'04*, pages 90–101, June 2004.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *SOSP '91*, pages 152–164, October 1991.

[7] M. Chaudhuri and M. Heinrich. SMTp: An Architecture for Next-generation Scalable Multi-threading. In *ISCA'04*, pages 124–135, June 2004.

[8] D. Chiou et al. StarT-NG: Delivering Seamless Parallel Computing. In *Euro-Par 1995*, pages 101–116, August 1995.

[9] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *HPCA-5*, pages 172–181, January 1999.

[10] M. Horowitz et al. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *ISCA'96*, pages 260–270, May 1996.

[11] InfiniBand Trade Association, InfiniBand Architecture Specification, Release 1.2, October 2004. Available from `http://www.infinibandta.org`.

[12] A. Jaleel and B. Jacob. In-Line Interrupt Handling for Software-Managed TLBs. In *ICCD 19*, pages 62–67, September 2001.

[13] N. Kirman et al. Checkpointed Early Load Retirement. In *HPCA-11*, pages 16–27, February 2005.

[14] K. Krewell. Sun's Niagara Begins CMT Flood: The Sun UltraSPARC T1 Processor Released. In *Microprocessor Report*, January 2006.

[15] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *ISCA'94*, pages 302–313, April 1994.

[16] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[17] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA'97*, pages 241–251, June 1997.

[18] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *PODC '86*, pages 229–239, August 1986.

[19] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[20] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *ICPP'95*, volume I, pages 1–10, August 1995.

[21] R. R. Oehler and R. D. Groves. IBM RISC System/6000 Processor Architecture. *IBM Journal of Research and Development*, pages 23–36, January 1990.

[22] K. Olukotun et al. The Case for a Single-Chip Multiprocessor. In *ASPLOS-VII*, pages 2–11. ACM Press, October 1996.

[23] X. Qiu and M. Dubois. Tolerating Late Memory Traps in ILP Processors. In *ISCA'99*, pages 76–87, May 1999.

[24] Z. Radović and E. Hagersten. Removing the Overhead from Software-Based Shared Memory. In *SC'01*, November 2001.

[25] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *ISCA'96*, pages 34–43, May 1996.

[26] D. J. Scales et al. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In *ASPLOS-VII*, pages 174–185, October 1996.

[27] I. Schoinas et al. Fine-grain Access Control for Distributed Shared Memory. In *ASPLOS-VI*, pages 297–306, October 1994.

[28] I. Schoinas et al. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *PACT'98*, pages 40–49, October 1998.

[29] S. J. Sistare and C. J. Jackson. Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect. In *SC'02*, November 2002.

[30] R. Stets et al. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *SOSP '97*, pages 170–183, October 1997.

[31] M. Tremblay et al. The MAJC Architecture: A Synthesis of Parallelism and Scalability. *IEEE Micro*, 20(6):12–25, nov 2000.

[32] D. Wallin et al. Vasa: A Simulator Infrastructure with Adjustable Fidelity. In *PDCS 2005*, November 2005.

[33] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, 2000.

[34] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA'95*, pages 24–36, June 1995.

[35] H. Zeffer et al. Exploiting Spatial Store Locality through Permission Caching in Software DSMs. In *Euro-Par 2004*, pages 551–560, August 2004.

[36] C. B. Zilles, J. S. Emer, and G. S. Sohi. The Use of Multithreading for Exception Handling. In *Proceedings of the 32nd IEEE/ACM International Symposium on Microarchitecture (MICRO-32)*, pages 219–229, November 1999.