Design and Performance of the Software-Controlled Coma

Adrian Moga

CENG 98-02

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4475
May 1998

DESIGN AND PERFORMANCE OF THE SOFTWARE-CONTROLLED COMA

by

Adrian Moga

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
Doctor of Philosophy
(ELECTRICAL ENGINEERING)

May, 1998

Dedicated to my dear family and my wonderful Nina.

Acknowledgments

I am finally at the end of a long road and I can look back to remember all the wonderful people that have made this possible.

I would like to thank my advisor for four years, Dr. Michel Dubois, for recognizing my passion for this topic and for offering me the chance to see my dreams come true. Dr. Shahram Gandeharizadeh and Dr. Timothy Mark Pinkston have been very kind to serve on my defense committee and I am grateful to them.

Alain Gefflaut has been my favorite research partner throughout the years and I thank him for the work we did together on the first paper on this topic and for the fruitful discussions we had ever since.

Lucille Stivers has always had lots of encouragements and good advice, for which I am grateful. I also thank her for her true compassion and help during difficult moments.

I have enjoyed the friendship and cooperation of many colleagues at USC. Thanks are due to Luiz Barroso, Jaeheonj Jeong, Kangwoo Lee, Koray Öner, Fong Pong, Xiaogang Qiu, Yong Ho Song, and Shi Zhi.

My many friends have made living in Los Angeles a pleasant experience. Thank you, Sasha Bucur, Diana and Radu Marculescu, Elena and Traian Mitrache.

Although far from my family, I was always encouraged by their belief in me. My mother, Stela, would have enjoyed this moment more than anyone. My father, Nicolae, has taught me to always do things right. Alina, my very smart sister, has always been the perfect combination of sense and sensibility. It is hard to imagine a more loving and kinder person than my grandmother, Eleonora Julei. Thank you all for understanding I had to fol-

low my call and for putting up with the long separation.

Last, but not least, I thank Nina Bahnean for her dear love. She has given me more reasons than one to see this work finished and she has always been on my side, patiently waiting for this moment. I hope I can return all the happiness you give me, my beloved Nina.

Contents

List of Figures

List of Tables

Abstract

Traditionally, cache coherence in multiprocessors has been maintained in hardware. However, the cost-effectiveness of hardware protocols for Distributed Shared Memory (DSM) systems is questionable. Virtual Shared Memory systems have highlighted the many advantages of software-implemented protocols, albeit at a performance price. The performance gap is narrowed by hybrid systems with software-implemented coherence protocols and hardware support for fine-grain access control.

This work contains the first proposal and evaluation of a hybrid COMA (Cache-Only Memory Architecture). The system is called SC-COMA for Software-Controlled COMA, to emphasize that the protocol engine is emulated by software executed on the main processor. Contrary to user-level protocols, the software handling coherence events in SC-COMA runs in sub-kernel mode, transparently and efficiently providing the same services to applications as a hardware counterpart. SC-COMA is employing a novel coherence protocol, optimized for a hybrid implementation, which has been fully implemented. The support for fine-grain access control is embedded in the memory controller.

The evaluation methodology is based on execution-driven simulation of complete applications from the SPLASH-2 suite. Our results show that SC-COMA is competitive and a viable solution to easily transform networks of workstations into powerful multiprocessors. On systems with 32 processors, it achieves a slowdown of 11-56% with respect to an aggressive hardware counterpart, across a range of applications and memory overhead. Scalability is good and faster processors favorably affect the performance. An investigation on the impact of memory organization on the performance of hybrid systems reveals

that, in most of a wide range of cases, COMA outperforms other alternatives: CC-NUMA, Simple COMA, and RC-NUMA due to the lower node miss ratio.

The performance of SC-COMA is further improved by three techniques: relaxed inclusion, mastership hints, and replacement hints. Even more significant improvements are obtained by adapting the SC-COMA approach to other hardware platforms: symmetric multiprocessor (SMP) nodes and processors with non-blocking stores.

Keywords: NUMA, Distributed Shared Memory, COMA Architecture, Software Coherence, Performance Evaluation

Chapter 1

INTRODUCTION

The raw processing power of computing systems is advancing at a fast and steady pace, in terms of both processor speed and system size for a given budget. To sustain efficient parallel and distributed processing, computer interconnection and network designers strive to achieve lower latencies and higher bandwidths. Computer architects bridge the gap between computation and communication speeds with advanced hierarchical memory designs and latency-tolerant techniques which overlap computation and communication. They also balance the entire computing system to avoid bottlenecks and to achieve scalability.

Software is the remaining key player in the complicated game of parallel processing. The shared-memory paradigm provides programmers with a global addressing space where a datum is uniformly referred to by its address. Thus, communication is natural and data movement is transparent. This is a great relief for programmers, but may come at a performance cost if the latencies of shared memory accesses are not accounted for. As it turns out, this is many times the case, a fact that puts a lot of pressure on the lower design levels: compilers, operating systems, and hardware.

Hardware solutions supporting the shared-memory paradigm have come a long way. The 80's bus-based designs have quickly run out of steam as the medium used for communication (the bus) could no longer support more than 16-32 bandwidth-hungry modern processors. Addressing the bandwidth and scalability problems, Distributed

Shared-Memory (DSM) systems partition the addressing space and assign these partitions to physically separate memory modules interconnected through a point-to-point network, as shown in Figure 1.1.

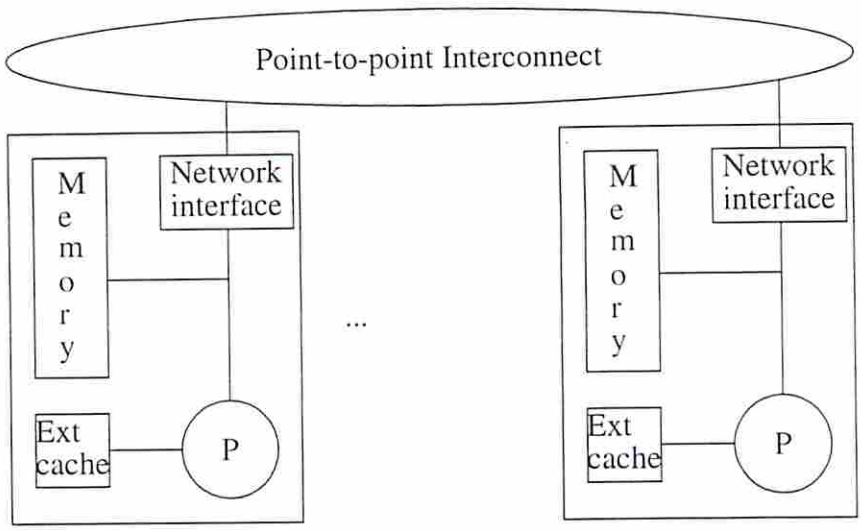


Figure 1.1. A distributed memory architecture

Regardless of the main memory organization, processor caches (usually, two or more levels) are an important source of performance gain by exploiting locality properties of programs. However, they introduce the problem of coherence, inherent when multiple copies of a datum are present in different physical locations. Essentially, enforcing the serialization of writes to the same address, coherence protocols provide the mechanisms by which processors are not allowed to read stale information. Usually, DSM systems are employing directory-based protocols [11] which are significantly more complex than the snoopy protocols [4] of bus-based systems.

The physical distribution of memory creates a Non-Uniform Memory Access (NUMA) architecture. A door is opened to optimizations by storing frequently used data in the local memory and accessing it (after a cache miss) with lesser penalties. Such an

opportunity can be exploited at several levels: application, compiler, operating system and hardware, being essentially a problem of data placement in the first three levels and data replication and migration in the last two. Clever initial placement controlled by the application or compiler [52], and page migration and replication by the operating system [79] can improve the situation, but there are limitations. Irregular sharing patterns and dynamic work scheduling defy static placement by applications and compilers, and un-padded or mixed read/write data can reduce the efficiency of data migration and replication in large, page-sized chunks by the operating system. The Cache-Only Memory Architecture (COMA) [35][10][40] provides automatic fine-grain data replication and migration at the main memory level, thus solving the NUMA problem, but unfortunately, further complicating the coherence protocol.

Several factors are motivating current research on DSM systems to investigate alternatives to hardware implementations. In spite of achieving the best performance, hardware DSM systems [12][48] are hard to build and costly, and lack flexibility. Ad-hoc hardware approaches also lengthen the system development time, a serious problem as processor speeds practically double every 18 months. All these disadvantages are eliminated by software DSM systems [50][2], at the cost of some performance degradation. This degradation is partly caused by false sharing [22] effects, due to the coarse units of coherence (the pages). Software-implemented protocols, however, can go to great lengths of complexity in order to improve protocol performance or hide remote access latency. The ever-increasing speed of processors drives the performance of software protocols even closer to hardware implementations [9]. Apart from the performance aspects, software DSM systems are better suited for off-the-shelf components and systems (networks of workstations [3]), which further improves their cost-effectiveness.

Hybrid hardware/software DSM systems provide cache coherence (not page-based) with the aid of simple hardware support for fine-grain memory access control and

software-implemented protocols. Thus, they preserve the advantages of software DSM, but further drive performance closer to hardware implementation levels.

1.1 Research Contributions

The research presented here contains the first proposal and evaluation of a hybrid cache-only memory architecture, thus with a software-implemented coherence protocol and assisted by simple hardware support for fine-grain access control. Some of the contributions of this work are:

- A novel coherence protocol for COMA. The protocol is optimized for software coherence by reducing the number of exchanged messages.
- A low-overhead design for integrating the coherence software into the system and a novel method for avoiding live-locks.
- The evaluation of the performance implications of a software-implemented COMA coherence protocol as compared to a hardware implementation.
- The evaluation of the impact of the memory organization on the performance and complexity of hybrid DSM systems.
- The proposal and evaluation of solutions that extend the hybrid COMA design to platforms of SMP clusters and to processor with non-blocking write operations.
- New results on several aspects of COMA architectures: data allocation, protocol optimizations based on hints, inclusion relaxation, and attraction memory parameters.
- The development of an efficient environment for the detailed simulation of a large number of hardware and hybrid DSM architectures.

1.2 Thesis Organization

The next chapter provides background information about software coherence, cache-only memory architectures (COMA), and performance evaluation using execution-driven simulation.

Chapter 3 presents the design of the software-controlled COMA (SC-COMA). It begins by describing the coherence protocol. The design of the hardware support for fine-grain access checking and the software implementation of the coherence protocol follow. The chapter ends with a description of the data placement strategies.

Chapter 4 is dedicated to the two-fold evaluation of SC-COMA's performance. The evaluation is preceded by a description of the methodology and the base architectural assumptions. First, to understand the performance costs of software coherence, SC-COMA is compared against an ideal hardware implementation. Latencies, execution times, speedups, and other factors are considered. Second, to investigate which memory organization is more advantageous for a hybrid implementation, we evaluate COMA against three other possibilities: CC-NUMA, with and without a remote data cache, and Simple COMA. Execution times and implementation complexity are compared.

Chapter 5 presents three optimizations that can improve the performance of SC-COMA. The relaxation of the inclusion property is beneficial for all applications and significant for some. Ownership hints can be successfully used in some applications to transform three-hop transactions into two-hop ones. Finally, two kinds of replacement hints are useful in reducing the number of replacement messages that are rejected and must be retried.

Chapter 6 presents two extensions of SC-COMA. First, we adapt the implementation of the coherence software to run on an SMP cluster. The performance benefits of using clusters instead on uniprocessor nodes are evaluated. Secondly, we upgrade the

coherence software for processors with store buffers in order to be able to take advantage of non-blocking writes and memory models other than sequential consistency.

Chapter 7 presents related work on COMA architectures, virtual shared memory systems, and hybrid distributed shared memory systems.

Chapter 8 contains the conclusions of this research and a discussion of future work.

Chapter 2

BACKGROUND

2.1 Software Coherence

2.1.1 The Coherency Problem

Memory and cache hierarchies are widespread in both uniprocessors and multiprocessors. Their effectiveness capitalizes on the locality of reference properties of applications, which allow the *working set* to accumulate in small and fast caches, close to the processor, while the bulk of the data set is stored in larger, slower memory levels. In multiprocessors, the working sets of several processors may overlap and, obviously, performance can be enhanced if each processor is allowed to cache copies of the needed data. The task of maintaining multiple copies of a datum consistent with each other in a memory system that allows replication is referred to as *coherency*. The complete set of rules by which data is kept coherent is defined by the coherence *protocol* [4][73].

Coherency is facilitated by the addition of explicit states to units of a given size, called the *coherence units*. When the coherence units are the size of cache blocks, coherence takes the form of *cache-coherence*. Page-sized coherence units are used by Virtual Shared Memory (VSM) systems [49]. A small coherence unit reduces the chances of false sharing [22] and cache fragmentation, but a large one can reduce cache miss ratios by prefetching effects. The optimal coherence unit size is application-dependent, 32-256 bytes in most cases, thus causing page-based coherence to suffer significant false sharing.

2.2 Mechanisms for Enforcing Coherency and Implementations

To complete a processor reference, the closest copy of the requested data must first be located in the memory hierarchy. When the reference is a read and the found copy is up-to-date (i.e. valid), the data is returned to the processor. Write references are more difficult to handle because writes must be serialized. Depending on whether the writer invalidates shared copies of the data, thus acquiring exclusive access, or updates the shared copies, the protocol is a *write-invalidate* or *write-update* protocol. To indicate that a datum is owned exclusively, and thus writable without further actions, an exclusive state is used. Particular implementations and optimizations might use other states, in addition to invalid, valid, and exclusive.

The process of locating a copy of the referenced data and checking its status, shown in Figure 2.1, is called *access lookup* [71]. Access lookup is performed on every shared-memory reference, thus a low-overhead implementation is important for high performance. Cache controllers routinely incorporate hardware support to perform access lookup, but software implementations are feasible [71][68].

Access lookup. For every shared reference:

- **Locate** (a copy of) the addressed item within the *local* caching space or identify its *remote* location.
- Access **verification** (valid, exclusive...).

For a miss only:

- Selective triggering of **protocol** actions.

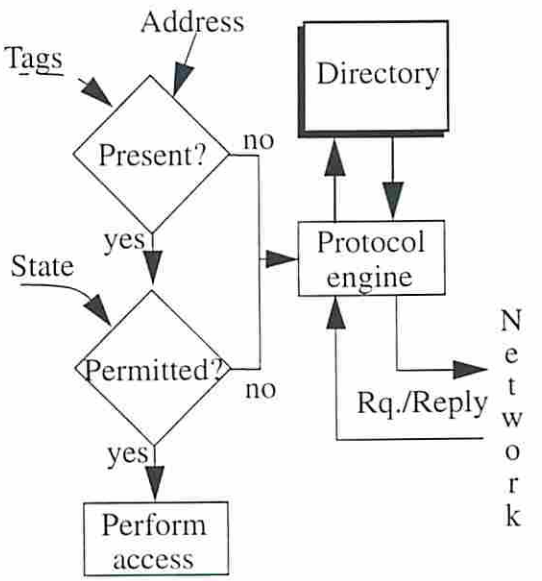


Figure 2.1. Mechanisms for enforcing coherence: access lookup and protocol engine

Failed access lookups trigger protocol actions which bring the requested data in the local caching space in the correct state. As shown in Figure 2.1, the protocol engine operation is based on the information stored in a directory. In a DSM system, protocol engines from all nodes cooperate in a request/reply manner, by exchanging messages over a point-to-point network. Thus, the protocol engine works both as a client, issuing requests to other nodes, and a server, handling external requests.

The protocol engine can be a custom hardware controller with low latency and occupancy, but software-implemented coherence can be supported with off-the-shelf microprocessors. When categorizing protocol engines, there is a gray area between hardware and software implementations as many controllers are programmable with ever increasing levels of flexibility. We draw the line for hardware-implemented coherence when the protocol engine is significantly different from a classic RISC microprocessor core.

Given the diversity of implementation opportunities for the access lookup and the protocol engine, it is no wonder that so many classes of DSM systems have emerged. Table 2.1 lists the four major classes. Obviously, there are many subclasses within each group based on other implementation choices.

DSM Class	Access Lookup	Protocol	Examples
Hardware	HW fine grain	HW	DASH, KSR-1, SGI Origin
VSM	HW page grain	SW	Ivy, Treadmarks
All-software	SW fine grain	SW	Blizzard-S, Shasta
Hybrid	HW fine grain	SW	Typhoon, FLASH, **SC-COMA**

Table 2.1. Implementations of access lookup and the protocol engine

Hardware DSM systems [48] use hardware support at all levels of the memory hierarchy to perform access lookup at the cache block level and a hardware controller as

the protocol engine. While performance, in terms of latency and occupancy of the protocol engine is high, so is the cost of design, development, and manufacturing. At the same time, flexibility and portability across platforms and generations is low.

Virtual Shared Memory systems [49] use the virtual memory support to perform access lookup at the granularity of a page. The coherence protocol is implemented in software executed at the user level on the main processor. VSM systems are inexpensive, very flexible, and portable. However, their performance is hampered by false sharing and the costly switching of the processor between application and protocol modes, intermediated by the operating system. Unless user-level network interfaces are available, the costs of messaging are considerable as well.

All-software DSM systems [71][68] replace the page-level access lookup of VSM with a fine-grain software lookup. For this purpose, application binaries are instrumented with checking code before the load/store instructions. This solution adds some overhead to the execution time but, compared to VSM systems, avoids problems caused by false sharing and the involvement of the operating system between a failed lookup and the execution of a protocol action. Additionally, it allows multiple grains of coherence for data.

Finally, hybrid DSM systems [62][44] are based on the observation that the complex and costly part of maintaining coherence is in the protocol engine, while the access lookup is simple, application-independent[1], and relatively inexpensive to implement even at the highest level of performance. Thus, they employ the same fine-grain hardware-implemented access lookup as hardware DSM systems do, but perform the coherence protocol in software, similar to VSM systems. In general, hybrid DSM distinguishes itself from VSM by more than just the fine-grain access lookup, as will be explained in the following section. Thus, hybrid systems enjoy the flexibility and low cost advantages of soft-

1. except for supporting multiple coherence grains.

ware DSM systems without the drawbacks caused by an access lookup that is either too coarse or incurs overheads.

SC-COMA is a hybrid DSM system using custom hardware support in the memory controller to perform low-overhead fine-grain access lookup beyond the cache level. The coherence protocol is executed in software on the main processor.

2.3 Software Implementation of Coherence Protocols

Using a microprocessor core to execute coherence protocol actions is possible in several ways. The most handy and inexpensive method is to multiplex the main processor between application and protocol modes, as done by VSM systems. Other solutions resort to the use of a dedicated protocol processor. Throughout the many possibilities, an important variable is the level of integration of the protocol engine with the other hardware and software components of a DSM node. A tight integration is somewhat in conflict with the cost incentives of using mass-produced workstations and SMP clusters as building blocks.

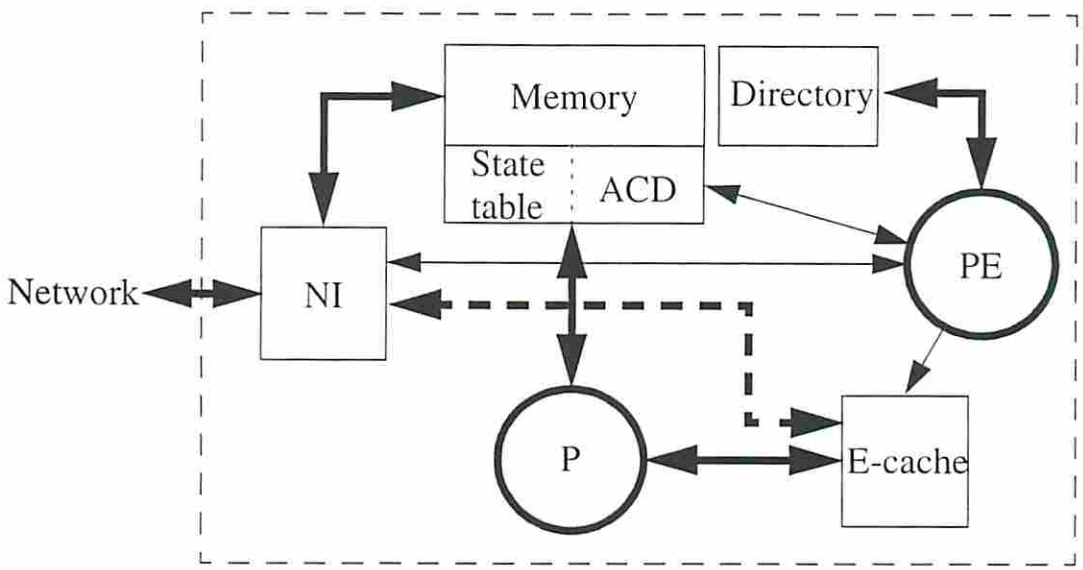


Figure 2.2. The main components of a DSM node

The protocol engine (PE) interacts with many components of a DSM node, as shown in Figure 2.2. The Access Checking Device (ACD), positioned on the path of processor P's memory references, triggers protocol actions and its state table is updated by the PE. On behalf of external requests, the PE instructs the cache hierarchy to downgrade (invalidate or make read-only) cache blocks. Very importantly, the network interface (NI) cooperates tightly with the PE, both notifying it of the reception of incoming messages and shipping the outgoing messages.

Using the main processor P as a protocol engine is likely to involve some overheads as the hardware layers (controllers and data paths) and software layers (kernel and drivers) separating P from the NI and the ACD are not optimized for low latency interaction. Furthermore, the cost of switching P between application and protocol can be important. VSM systems are hardest hit, mostly by the costly traversals of system software layers required by interaction paths that are compatible with shrink-wrapped operating systems. By contrast, the integration of coherence software between the hardware and the kernel, although non-standard and slightly less flexible, is a much lower overhead solution.

A dedicated protocol processor can be integrated at different degrees in a node. At the low end, one of the processors in an SMP cluster can be used for this purpose [63][16]. In addition to off-loading the application processor(s), the protocol processor does not need to switch context and can respond to events faster, by using polling. At another level, the protocol processor is integrated with the NI and the ACD, as a bus snooper [61]. Private data paths provide the extra benefit of faster communication between components. At the highest end, a node controller including the protocol processor interfaces directly to the memory, cache, and I/O, providing pipelined data paths [44].

2.4 Cache-Only Memory Architectures

Cache-only memory architectures (COMA) address a limitation of the cache-coherent non-uniform memory architectures (CC-NUMA): data can only be replicated at the cache level. In situations where the application's working set is large and does not fit in the caches, thus when the cache capacity miss ratio is large, main memory is accessed frequently. Accesses to remote main memory modules are especially costly and degrade performance by causing significant processor stalls. This makes CC-NUMA very sensitive to the placement of data in main memory. Techniques addressing the problem of data placement in CC-NUMA have limitations and require either cooperation between the programmer and operating system [79] or a dedicated cache for remote data [48].

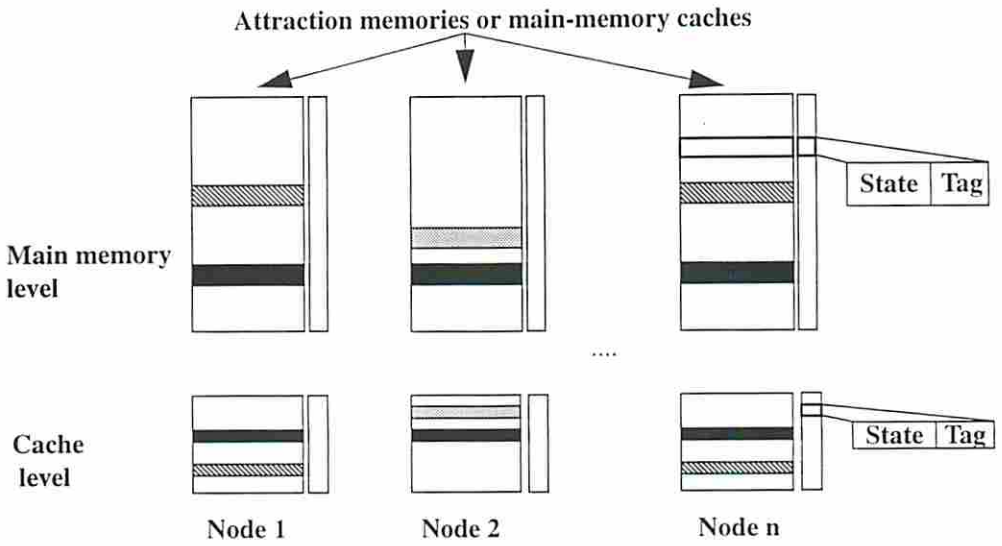


Figure 2.3. The cache-memory hierarchy in COMA

COMA overcomes the replication limitation of CC-NUMA by organizing the main memory of each node as a large set-associative cache, called *attraction memory*, shown in Figure 2.3. The attraction memory allows automatic data replication and migration beyond the cache level and can capture very large working sets. The attraction memory

blocks, which are also the inter-node coherence units, are called *lines* to distinguish them from processor cache blocks.

The coherence protocol for COMA faces a unique problem caused by the lack of a memory level below the attraction memories. Thus, when a line is victimized by an attraction memory, the node must first ensure that copies of the line exist in other nodes. For this purpose, each line has a designated master node. Lines can be victimized silently by non-master nodes, but a master node must generate a *replacement* request to inject the line into another attraction memory.

COMA has a memory overhead. To allow room for replication, the allocated memory must not exceed the combined size of the attraction memories. The ratio between these two measures is called *memory pressure*. A high memory pressure (little room for replication) will reduce the attraction memory hit ratio and increase the rate of replacements, thus degrading performance. For most applications, a memory pressure of 75% or less is acceptable.

2.5 Execution-Driven Simulation

The numerous and complex factors of performance in modern computer architecture make evaluation a very difficult task. The variability exhibited by application workloads usually makes analytic modelling inaccurate and good as a first-order approximation at best. Hardware prototyping, although very accurate, is expensive and lacks the flexibility one might desire in the first stages of design and experimentation. Furthermore, the fast pace of change in processor technology and architecture oftentimes forces a design to be ready simultaneously with the processor being available.

Simulation techniques, although slow and less accurate not long ago, are making steady progress, fueled also by ever faster processors and larger caches and memories. At this point, one can talk about the beginning of virtual machines, such as SimOS [65]. Exe-

cution-driven simulators contain a core for the functional simulation of applications. This core is intimately connected to a certain processor architecture (i.e. instruction set) and more or less acurate in the modelling of internal processor activity. The functional simulator interacts synchronously and on-line with detailed architectural simulators for the memory subsystem, interconnect, and I/O, by feeding them with memory references and stalling as necessary. All the events are time-stamped and the latencies, and hand-shaking and arbitration between the modules are modeled in detail. We base our evaluations on such an execution-driven simulator described in more detail later.

Performance evaluation has been helped tremendously by the emergence of standard sets of benchmarks, such as SPLASH and SPLASH-2 [82], SPEC95, and NAS. They constitute a common measure of performance for existing architectures and new proposals and allow a more realistic way to evaluate the cost-effectiveness of systems. In our evaluations, we use benchmarks from the SPLASH-2 set.

Chapter 3

THE DESIGN OF SC-COMA

3.1 The Coherence Protocol

SC-COMA uses a centralized, write-invalidate coherence protocol. The protocol combines features found in the DASH [48] multiprocessor and the COMA-F [40] proposal with novel solutions designed to enhance the performance of a software implementation.

Like DASH and COMA-F, SC-COMA uses a flat directory structure. This is in contrast to the hierachical directory scheme proposed for DDM [35]. Our intuition is that a software implementation would only amplify the higher latency shortcomings of hierarchical directories as compared to flat structures [74]. Consequently, every memory line in SC-COMA has a statically-associated *home* node. Every attraction memory miss is handled by first sending a request to the home node. Traditionally, this is where the directory information (copyset, state, etc.) for the line is stored. In SC-COMA, however, the only attribution of the home node is to serialize incoming requests and to forward them to the line's current master (i.e. the last writer). Other directory information, such as the copyset (i.e. the presence information), migrates to the current master along with the data. This allows an efficient implementation of a non-blocking directory and the minimization of the number of exchanged messages and of the protocol engine occupancy.

Next, we will describe the protocol data structures, the protocol transactions, the novel features of SC-COMA's protocol, and the replacement algorithm.

3.1.1 Coherence-Enforcement Data Structures

In order to maintain multiple copies of a datum coherent, a COMA needs to record and utilize data specific to both caches (tags and states) and other directory-based NUMA systems (such as the copyset). These data, along with the set-associative attraction memories, are shown in Figure 3.1 and will be described in detail next.

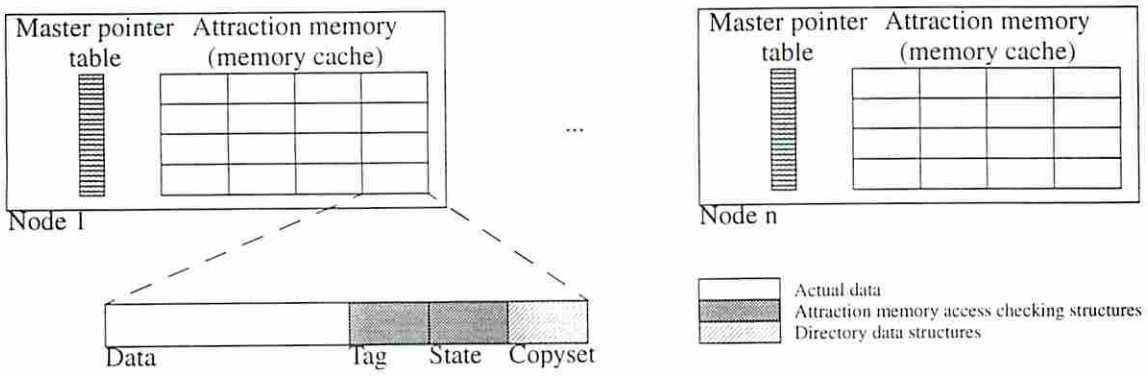


Figure 3.1. Attraction memories and coherence-enforcement data structures

The **tag** identifies a memory line stored in a frame of an attraction memory, thus allowing data to freely migrate and replicate in main memory. Because attraction memories are large, tags are short: only $\log(an)$ bits are required in a system with n nodes and for an attraction memory associativity of a. In the base configuration of SC-COMA, n=32 and a=4, thus tags are seven bits long. However, a whole byte is used for storage to simplify their manipulation by general-purpose processors.

Data cached in the attraction memory of SC-COMA can be in one of four stable **states**, similar to COMA-F. These states are Invalid, Shared, Master-Shared, and Exclusive. The last two states denote mastership of the memory line; before the frame of a master copy can be reused in the attraction memory, a node must first generate a replacement message, as will be discussed later. There are also several transient states for pending transactions. The encoding of the state could take as few as three bits, but again SC-COMA uses a whole byte. It is convenient to combine and pack the tag and state storages,

so that the information for all the frames in a set can be fetched in a single access whenever the attraction memory must be searched. For the base configuration of SC-COMA, eight bytes (a double word) store the combined tags and states for a set.

The **copyset** is a full bitmap representation of the presence information. It is used to send point-to-point invalidations of multiple shared copies of a line when a node is requesting exclusive access. In most protocols, the copyset is maintained by the home node, which is also responsible to send out the invalidations. As will be shown later, the decision to maintain the copyset at the current master in SC-COMA supports the implementation of a non-blocking directory through request buffering, which minimizes the number of exchanged messages. It also provides implicit replacement hints for lines victimized in state Master-Shared. There are also disadvantages to this decision: a higher memory overhead and the necessity to migrate the copyset.

Home-based copysets are maintained for every physical memory line; in SC-COMA, copyset storage must be associated with every attraction memory frame. To allow data replication, the number of attraction memory frames is larger than the number of physical memory lines. However, due to the small size of copysets, their overhead is negligible compared to the overhead for data memory. Furthermore, SC-COMA's copyset storage is multi-functional and supports some protocol optimizations. A copyset needs to be associated with a frame only in the Master-Shared state. As will be shown later, the copyset storage can be used to store mastership hints for frames in states Invalid and Shared or replacement hints for frames in state Exclusive. Mastership hints allow two-hop transactions by bypassing the home node; replacement hints increase the chance that a replacement message is accepted at its first destination.

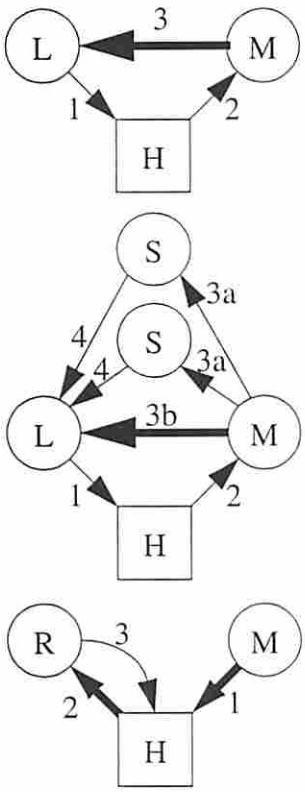
In SC-COMA, the copyset must always follow the associated line when the line changes mastership, which means it must be included in the message. The only such necessary situation is the replacement of Master-Shared lines, which is not so frequent. When

mastership changes due to a node acquiring an Exclusive copy, only the count of bits set in the copyset must be communicated to allow the requestor to collect invalidations. The copyset at the new master is set to zero. However, to simplify the software implementation in SC-COMA, the whole copyset is sent. In some protocols [40][85], the mastership of a shared line is passed on to the last reader, which would significantly increase the number of cases when the copyset needs to migrate in SC-COMA. Overall, the migration of copysets is less of a performance problem and more of a scalability limitation. The current implementation of SC-COMA supports 32 nodes (four bytes for a copyset). With four-way SMP nodes, a total of 128 processors can be connected, which is quite satisfactory.

Master pointers for every physical memory line are stored in a table at the home node. The table has one byte entries and it is indexed by the physical line address after masking off the home node bits. Requests received by the home node are simply forwarded to the current master. The rule in SC-COMA's protocol is that the master of a line is the last writer or a node which accepts the line following a replacement. To handle some very infrequent corner situations, the master pointer can have a special "locked" value, which forces the home node to send negative acknowledgments (NACK) to any requests. Another special value for the pointer is used to support efficient first-touch data allocation. Pages that are zeroed on demand (as many are in the shared data segment) are not allocated at their home node immediately after the page fault. Rather, master pointers for its lines are set to the "not allocated" value. A line will be allocated in the attraction memory of the first node to reference it after the home node responds to the miss request with a special reply containing no data. This also reduces memory fragmentation until the first time the page is swapped out.

3.1.2 Protocol Transactions

Figure 3.2 illustrates the flows of protocol transactions assuming that the home node is never identical to the local node (which experiences the miss) or the master node. In case the home node is one of these two nodes, no forwarding of requests is needed and the transaction completes in just two hops.



Read request

1. Local node (L) sends read request to home (H) node.
2. Home node forwards read request to current master (M) node.
3. Master node replies with data to local node. Master node state becomes (or remains) master-shared: local node state is set to shared.

Write (exclusive) request

1. Local node (L) sends write request to home (H) node.
2. Home node forwards write request to current master (M) node.

3a. Master node invalidates other copies at sharing (S) nodes (not needed if master has exclusive copy).

3b. Master node replies with data and copyset (no data is sent if local node only requests exclusive access).

Master node and other sharing nodes' state becomes invalid: local node state is set to exclusive.

Replacement request

1. Master node (M) sends replacement request (data and copyset) to home (H) node.
2. If home node cannot accept the replacement, it forwards it to a remote (R) node.
3. Remote node sends acknowledgment to home node. Remote node state is set to exclusive (if copyset is null) or master-shared (if copyset is not null).

Figure 3.2. Coherence protocol request flows

Read requests are sent to the home node. The home node consults the master pointer for the line and forwards the request to the node indicated by the pointer, which is the current master node. The master node sends the line directly to the requestor, updates the copyset and sets the line state to master-shared (possibly downgrading from exclusive,

in which case the processor cache blocks must be downgraded to read-only as well). After receiving the line, the local node sets its state to shared.

Write requests are handled similarly to read requests by sending the message first to the home node, which then forwards it to the current master. In addition, the home node also updates the master pointer with the identifier of the local node (the last writer). Subsequent requests forwarded by the home node to the local node might have to be buffered in case the current transaction is not completed. This feature of the protocol, called request buffering, will be described later. Upon receiving the request, the current master issues invalidations of other shared copies of the line (if any) and replies to the local node with the data and the copyset. The entire copyset is sent to the local node to avoid the costly counting in software of the copyset population. This number is needed to let the local node know how many invalidation acknowledgments to collect.

Exclusive (write-on-clean) requests are handled just like write requests except the current master replies to the local node with the copyset only. An interesting optimization is possible considering that in the overwhelming number of cases just one invalidation must be performed [34]. Instead of the master node sending the copyset to the local node, the copyset could be included in the invalidation message and passed on to the local node via the invalidation acknowledgment. Just two messages are necessary to deal with most exclusive requests. The benefits are lower latency (local node must process just one message instead of two) and reduced overhead at the master and local nodes (although the overhead increases slightly at the sharing node).

Replacement requests are always sent to the home node first[1]. If the home node cannot inject the line into its attraction memory, it locks the master pointer and forwards the replacement to a remote node. The process of forwarding continues until one of the

1. This requirement will be removed later when replacement hints are considered.

nodes finds room in its attraction memory and sends an acknowledgment back to home. The home node then updates the master pointer. While the master pointer is locked, every request is rejected by sending a NACK message which forces a retry.

3.1.3 Request Buffering

In a manner similar to the DASH protocol [48], SC-COMA uses reply forwarding to reduce miss latency and, very important in a software implementation, protocol engine occupancy. As such, a request received by the home node is forwarded to the current master which responds directly to the local node (we mention here that whereas this request flow applies only to "dirty-remote" blocks in CC-NUMA, it is very common in COMA for shared lines as well). Some questions arise when write/exclusive requests are forwarded: when to change the mastership at the directory and how to treat other requests while this change is pending.

The solution in DASH is to have the current master send an explicit mastership update message to the home node and to keep forwarding other requests to the current master until this update message is received. The disadvantages of this solution are the extra message and the fact that requests forwarded before the mastership update will be NACKed as their destination has just given up mastership. With a software implementation, both the weight of the extra message and the window of time during which an mastership change is pending and forwarded requests might get NACKed are significant.

SC-COMA's solution is to update the master pointer immediately after a write/exclusive requests and to forward other requests to this new master. This requires buffering of forwarded requests at the new master in case they reach it before the reply to the new master's pending request. In the general case, as shown in Figure 3.3, a list of pending requests will be generated and distributed between the buffers created at each node that has a pending write. After receiving the reply to a write request and performing the write,

every node checks for and replies to buffered requests. This is very convenient in a software implementation as the overhead of replying to buffered requests is compacted in a single necessary handler.

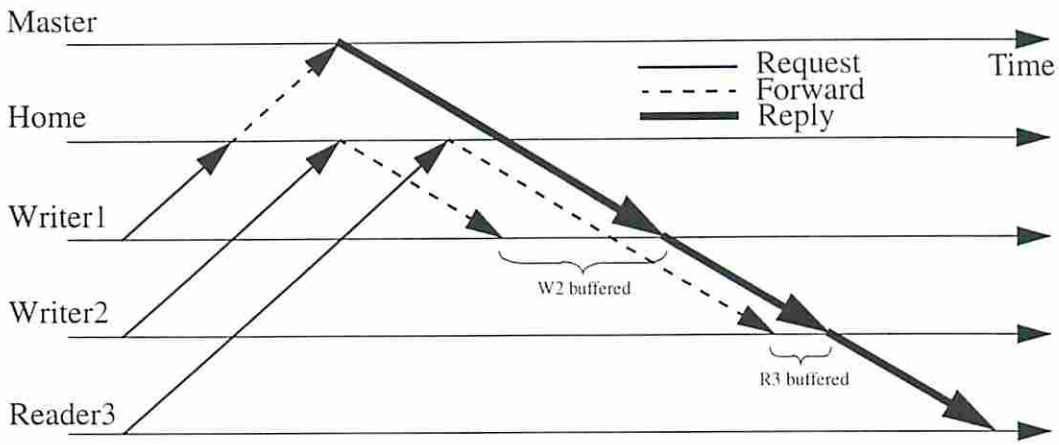


Figure 3.3. An illustration of request buffering

As long as a node does not give up mastership voluntarily (because of replacements), request buffering is obviously free of deadlocks and livelocks. We will now describe how to handle situations whereby the home node is forwarding a request to an master that has just performed a replacement and, thus, is unable to reply. The situation is slightly complicated by the possibility that a distributed list of buffered requests might have been created, as shown in Figure 3.3.

Read requests reaching a stale master are simply answered with a NACK sent to the local node. A read request does not change the master pointer at the home node and no other requests must be buffered at the local node, thus the request can be reissued without any complications. Because such events are quite rare, there is no need for optimizations.

Write requests are handled differently, as shown in Figure 3.4. A NACK message is sent by the stale master to the home node instead of the local node. As described before, the replacement request was also sent to the home node. Had the home node been able to

detect the crossover between the replacement and write requests and also to identify the node originating the write request, it could simply reply to the write request as if no replacement ever took place. The crossover is easy to detect at the home node: there is a mismatch between the sender of the replacement request (the former master) and the current master pointer. Unfortunately, the master pointer can be overwritten several times by multiple write requests and the home node cannot identify which node's write came first and had its request misdirected to the stale master. This is precisely why the NACK containing the identity of this node is sent to the home node. Thus, the replacement request is buffered at the home node until the NACK is received at which point the home node replies with the buffered line[1]. Meanwhile, other requests received at the home node are forwarded and buffered at the last writer, as usually.

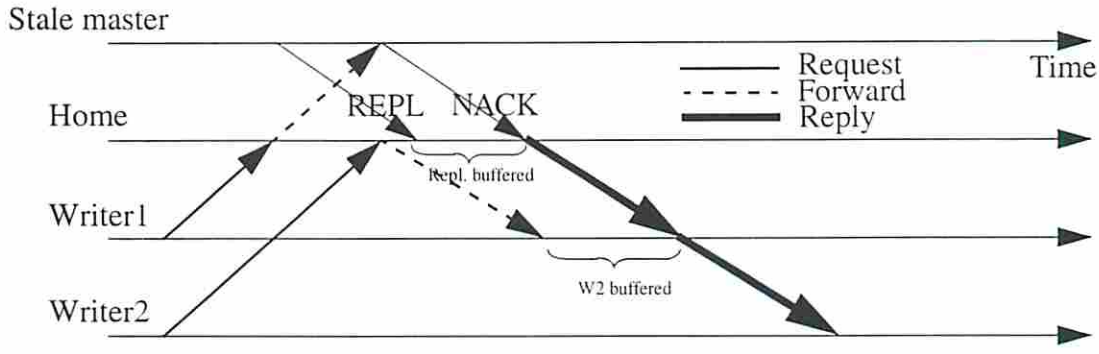


Figure 3.4. Crossover between replacements and write requests

The home node's buffer for crossover replacements can be very small, just one or two entries, because these events are very rare. In case of a buffer overflow, the home node can simply handle the replacement as usually. When the NACK is later received at the home node, the NACK is forwarded along the list of buffered requests to force a retry at each node. As a general rule, in case the forwarded replacement is accepted at a node

1. We are assuming that requests are delivered in order for point-to-point connections.

which already has a pending request (and is potentially on the list of buffered requests), the acceptor must wait for the NACK before it can acknowledge the replacement and resume its pending access.

There is one subtle point which further complicates the problem of crossover replacements. We have assumed that a stale master, upon receiving a request and not finding the line in its attraction memory, sends a NACK to either the local or the home node. However, if the stale master experiences a write miss right after replacing the line (see Figure 3.5), it will attempt to buffer incoming requests, as it is normally done. The situation is detected at the home node when the write request from the stale master is received and the line is found in the buffer for crossover replacements[1]. The line is then sent to the stale master, as it is the head of all buffered requests. The master pointer remains unchanged because it correctly indicates the last writer in the order requests have been buffered.

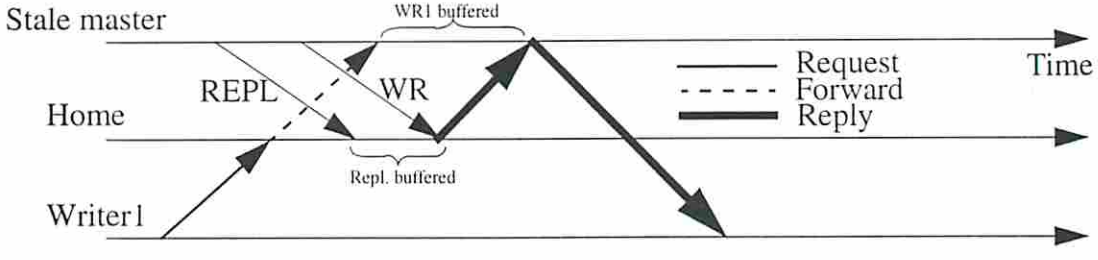


Figure 3.5. A special case of race in request buffering

3.1.4 Support for Efficient Synchronization

SC-COMA's coherence protocol supports efficient synchronization in the form of queue-based locks. Heavily contended locks generate a lot of traffic and synchronization penalties under the common test-and-test&set implementation. The penalties are expected to increase significantly with software-implemented coherence.

1. Again, we assume that the point-to-point connection preserves the order between the replacement and write requests.

Lock acquire and release operations are implemented with special atomic test&set and swap instructions. Failed test&sets trigger a special acquire protocol request. Acquire requests are sent to the home node which processes them as usual write requests, thus updating the master pointer. Acquire messages are then forwarded to the current (or future) lock master. If the lock is busy (or the acquire is pending), the request is buffered. A single-entry buffer per lock is sufficient for this purpose. The lock and the buffer are stored on the same memory line which cannot be shared by any other application data. Reply messages to an acquire do not cary any data, further improving the performance of synchronization.

When acquires are buffered, it is important to make the upcoming release (swap) visible (i.e. not allow it to complete in the cache) so that a software action can be triggered and pass on the lock. Declaring all locks as uncacheable is an easy solution, but less efficient in the absence of contention. Our solution is to always flush the block from the cache before buffering an acquire of a busy lock (or right after a successful acquire when a request has already been buffered). One must also make sure that there are no intervening accesses to the block between such a flush and the expected lock release, as they would cache the block again. This corresponds to the "normal" use of locks, but does not guard against programming errors. One way to catch erroneous use of locks is to add a special state for blocks with buffered acquires and monitor cache write-backs (they should not occur in this state). Another possibility is to add time-out to lock acquires. Finally, a simple, yet less transparent solution is to add explicit code to the lock release routine to check if an acquire is buffered and forcing a flush is necessary.

3.1.5 The Replacement Algorithm

The replacement algorithm comprises two policies: the victimization policy and the injection policy. The victimization policy decides *which* of the frames in a set of a

given node's attraction memory should be reused when room must be made to accommodate a new line. When the node has mastership of the victimized line it must send a replacement message such that the line will be injected into the attraction memory of another node. The injection policy decides upon the node(s) *where* the replacement is sent.

SC-COMA's victimization policy uses a priority scheme based on the line's state as follows:

1. Invalid
2. Shared
3. Master-Shared or Exclusive

This policy aims to reduce the number of replacements by delaying the victimization of lines in master mode but it does not always lead to an optimal attraction memory hit ratio. It is, however, one of the best policies that can be used when no LRU information is available. It is important in cases 2 and 3 above to break ties (i.e. multiple choices) in a randomized manner and not just to scan the set in some predefined order until a candidate is found.

The injection policy used in the early version of SC-COMA was to send the replacement message to the line's home node first. If not accepted at node n, a replacement was forwarded to node $(n+1)$ *modulo* N, where N is the number of nodes. This policy is flawed and we will later show why and how to improve it. We will also show how replacement hints can be used to increase the chances that a replacement is accepted at its first destination, thus reducing the overhead and traffic associated with replacement forwarding.

3.2 Shared-Memory Fine-Grain Access Checking

Like hardware NUMA systems, SC-COMA performs fine-grain shared-memory access checking at all levels of the memory hierarchy: processor caches and main memory cache

(attraction memory). Access checking in the processor caches is a common feature of all multiprocessor systems and it is incorporated in a standardized manner in the design of microprocessors and chip sets. There is, however, a large range of mostly ad-hoc methods of performing access checking for cache misses. SC-COMA implements access checking in the memory controller. To support fine-grain sharing, access checking in SC-COMA is based on a table of address tags and state associated with each shared-memory line; this table is stored in main memory.

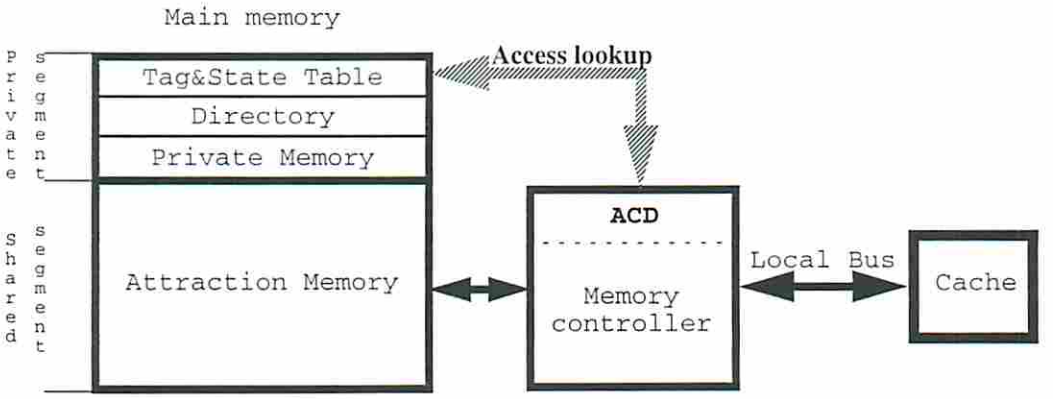


Figure 3.6. The Access Checking Device and main memory partitions

Cache misses and write-backs are presented to the memory controller, usually over an arbitrated local bus. For most systems, the bus address is identical to the physical memory address. However, in COMA, main memory is organized as a set-associative cache and a tag-based search is required before the correct data can be fetched or written back (assuming the memory cache is not direct-mapped). To minimize the overhead of tags in terms of both space (storage) and time (checking), we partition the main memory into private (tag-free) and shared segments, as shown in Figure 3.6. The private segment stores the stack, read-only data and code segments replicated under operating system control, and other information that does not need to be shared, such as the directory. For a low-cost

implementation of SC-COMA, as will be assumed throughout, the private segment also stores the tag and state table. The shared segment constitutes the attraction memory which supports automatic data migration and replication.

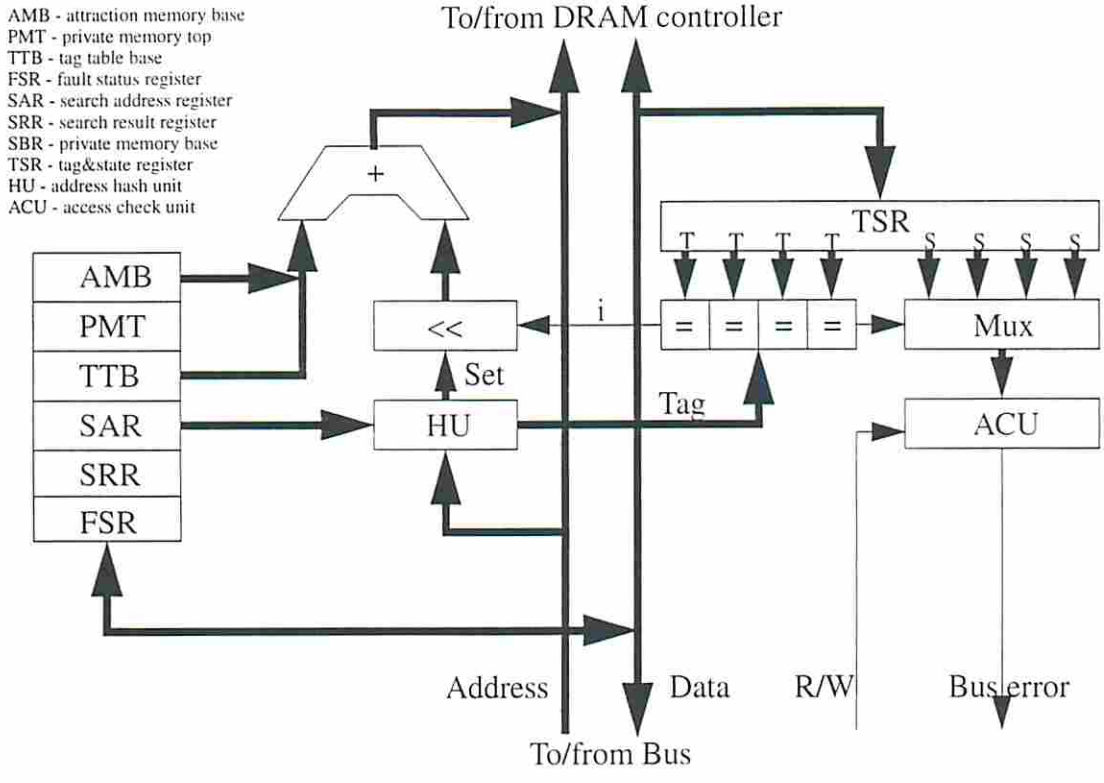


Figure 3.7. Internal organization of the Access Checking Device

Figure 3.7 presents a detailed view of the ACD. The address and data lines cross over from the bus to the DRAM controller. Bus addresses in a range from zero to the value indicated by the Private Memory Top (PMT) register are passed on unmodified to the DRAM.

Bus addresses exceeding the value in the Attraction Memory Base (AMB) require the ACD to first synthesize an address for the tag&state information. The Hash Unit (HU) splits the bus address into a Set and a Tag. The Set is appropriately shifted and added to the value in the Tag Table Base (TTB) register. Tag and state information for all the frames in

the same set is fetched into the TSR in a single DRAM access. The tags in the TSR are compared associatively against the Tag coming from HU. The outcome of this tag matching is used to multiplex the appropriate state information and forward it to the Access Checking Unit (ACU). In the event of a tag mismatch or access to an invalid line or write to a read-only line, we have an attraction memory miss. The ACD stores information (address, access type, outcome of tag comparison) into the Fault Status Register (FSR) and signals the miss by asserting the Bus Error line.

Attraction memory hits are followed by the actual data access. The tag comparison outcome, indicating which of the frames in the set must be accessed, is appropriately shifted and added to the AMB value. The resulting address is again passed on to the DRAM for a block transfer.

Often, during the handling of external requests, the attraction memory must be searched for a given line. The software handlers could inspect the Tag&State Table, but this is too slow and specialized hardware is already included in the ACD. For searching purposes, we added two registers: Search Address Register (SAR) and Search Result Register (SRR). When written, the SAR triggers the same operations as in a usual access, only the outcome is stored in the SRR register for inspection by the software handlers (a single SWAP instruction is sufficient to write the SAR and read SRR).

The attraction memory must support an access mode where access (state) checking is disabled. This is necessary for line fills and reads from the network interface. A shadow address space can be used for this purpose. For diagnostic reasons, the attraction memory can also be accessed in a direct mode (bypassing tag checking) in an address range contiguous to the private memory range and running up to the local memory size limit.

The maximum associativity of the attraction memory is limited by the memory width, which dictates the number of tag&state pairs that can be fetched in a single memory access. This, however, is not a serious limitation as the tags and state for a set of an eight-

way set-associative attraction memory can be packed into 16 bytes, which is a very reasonable memory width.

Some considerations about the tag-checking overhead in COMA. SC-COMA's solution to store the tag table in main memory, thus incurring the penalty of an extra access to fetch the tags, is an ad-hoc inexpensive method. Several solutions are possible to reduce or eliminate this penalty. Taking advantage of the fast page-mode access can reduce the penalty of fetching data after the tag from the same DRAM page (as done in S3.mp [57]). The use of dedicated SRAM memory modules for the tag table can make the outcome of the tag comparison available before the data DRAM enters the CAS cycle. However, it is very likely that, if COMA gains acceptance, specialized DRAM chips combining the data array for a wide set-associative memory and the tag storage will become available. There is considerable bandwidth inside the chip to simultaneously fetch all the blocks for a set with 8-16 frames and multiplex the appropriate frame based on the on-chip tag comparison. Still, many issues remain to be investigated, such as how to optimize the access to the data array.

Another issue concerns the partitioning of main memory into private and shared segments. To avoid the lack of flexibility of fixed partitioning, the operating system could attach a private/shared attribute to pages. This attribute would be stored in the TLB and carried over onto a special bus signal during cache misses, so that it becomes visible to the ACD. This scheme avoids the overhead of checking tags, but the size of tag storage increases, as any page can be part of the attraction memory.

3.3 The Software Coherence Handlers

The following describes the software shared-memory emulation layer of SC-COMA. We present the mechanisms required to perform software coherence actions on the application processor and the method for correct and efficient integration of the coherence handlers into the system. This description assumes a processor with blocking loads and stores.

Later, we will describe modifications required to support non-blocking stores. Unlike VSM systems or the Simple COMA [67], the software coherence protocol in SC-COMA uses physical identifiers for the coherence units, just as hardware DSM systems do.

3.3.1 Mechanisms for Software Coherence in SC-COMA

The implementation of transparent software cache-coherency on the application processor requires three basic mechanisms. The first is the ability to *trap* the processor in the event of an attraction memory miss and start an execution thread which brings the desired data in the correct state into the local memory. The thread gaining control in this event implements the client aspect of coherency protocols, thus initiating transactions. The second required mechanism is the ability for any two nodes in the system to asynchronously exchange messages. The asynchronous reception of a request *interrupts* the application processor and starts an execution thread implementing the server aspect of the coherency protocol. Finally, on behalf of nodes requesting exclusive or shared access to a datum, the processor must downgrade the status of coherence units at all levels of the cache-memory hierarchy. This requires some non-standard software control over the processor caches.

The Attraction Memory Miss Handler

Attraction memory misses detected by the ACD are signaled using the Bus Error signal. The assertion of this signal reaches the processor via the Memory Exception (MEXC) pin. Based on our assumption of blocking load/store operations, a memory exception triggers a *synchronous* access fault handler: when the fault occurs, the program counter is that for the faulted instruction. Thus, all instructions preceding the faulted load/store are completed, and the faulted instruction and all that follow are cancelled from their partially executed stages as if they never got started. This feature is also referred to as *pre-*

cise exception. Upon a precise exception, a new thread (the exception handler) starts to execute; at its termination, the faulted thread resumes by re-executing the faulted instruction (unless otherwise arranged by the exception handler).

Non-blocking load/store operations, when faulted, generate *asynchronous* exceptions: after the load/store is issued, other instructions are allowed to enter the execution stage and possibly complete. Thus, the program counter at the time of the fault is not directly correlated with the faulting instruction. Furthermore, when intermediate instructions are allowed to complete, restarting the program at the faulted instruction is not possible because some instructions would execute twice. Because most systems view memory exceptions as uncommon, rather catastrophic events (hence, the exceptional attribute), and because it optimizes their design, processors do allow instructions following a non-blocking store to complete as soon as the store is deposited into the store buffer. As will be described later, even stores faulted asynchronously in the store buffer can be brought to successful completion, assuming the store buffer is completely visible to the software.

After getting started, the attraction memory miss (or MEXC) handler goes through three phases. First, it consults the FSR in the ACD to find out the faulted address and access type. Based on this information it issues a request to a remote node. Once the request has been issued, the handler allocates a frame in the attraction memory for the incoming line, if necessary. If the set is full and all the frames are in master mode, a replacement request must be issued as well. To be noted that the allocation in the attraction memory and the issue of replacement requests is not on the critical path of a miss, and that we are utilizing idle processor cycles for this purpose.

After the first phase, which runs in non-interruptible mode, is done, the MEXC handler enters the second phase, waiting for a reply. In order to allow the processor to service external coherence requests and to avoid deadlocks, the processor must be interrupt-

ible during this phase (alternatively, it could poll on the network interface reception buffer).

After a reply is received and processed, the MEXC handler enters its final phase. This could be as simple as returning to the application and restarting the faulted instruction. However, to avoid possible livelock scenarios and to support the request buffering feature of the coherence protocol, this third phase is more complex, as will be described later.

The Message Interrupt Handler

The reception of a message (request or reply) in the network interface generates an asynchronous processor interrupt. This interrupt starts the execution of a message handler (the CINT handler) which runs in a non-interruptible manner. If the message is a request, the handler treats the request and sends a reply (or just forwards the request), and terminates. When the message is a reply, the handler fills the attraction memory frame allocated during the first phase of the MEXC handler with the incoming data from the network buffers (for mastership replies and invalidation acknowledgments, no data transfer is necessary). A reply handler also sets up the conditions for the MEXC thread to recognize that it can enter in its third phase and terminate.

Software-Controlled Downgrading of Cache Blocks

External coherence requests, processed by the CINT handler, might ask a node to give up exclusive access to a data (sharing request) or give up the access rights entirely (exclusive request). The processor has full access to the tag and state table and, thus, can downgrade any line in the attraction memory by a simple store instruction. To maintain inclusion, the state of the cache blocks must also be downgraded correspondingly.

Downgrading individual cache blocks, based on their physical address, is not a common feature in many systems. In some cases, the cache tags can be accessed in special diagnostic modes. For direct-mapped caches, a possible trick is to force a conflict in the secondary cache. This, however, is both inefficient and lacking generality. A more general solution is to force, under software control, a bus request that achieves the required downgrade. The ACD could be enhanced to support such programmable downgrades.

We have assumed that the processor does have means to directly control the cache tags by issuing two instructions: invalidate and flush (downgrade to read-only). For our SPARC-based platform these two instructions are implemented as loads from the untranslated physical block address in special alternate spaces [76]. These alternate spaces are recognized by the cache controller which carries out the downgrade autonomously.

3.3.2 System Integration of the Coherence Handlers

Sub-Kernel Coherence Handlers

Implementing shared-memory in software raises the problem of choosing the level where the coherence protocol should be integrated. The protocols in virtual shared memory systems and some hybrid DSMs are implemented at the user level (Figure 3.8 (a)) with the operating system providing basic services, such as memory allocation, servicing of messages, and handling of events generated by the hardware level.

In user-level shared-memory local memory misses are first handled by the operating system, which finally generates a signal to the application layer. The latency of such signals is considerable for most commercial operating systems (thousands of instructions [75]). Some researchers have pointed out more efficient methods of passing (memory-related) synchronous exceptions to the user level by modifying the kernel [75]. Still, some user state must be saved and the latency is still that of hundreds of instructions. Finally,

processors with special support for memory-informing operations [38] can invoke user-level routines on every cache miss. This is a very powerful mechanism but bound to have some overheads, as not every cache miss requires software intervention.

Unless the system provides a user-level network interface (such as CM-5's [71] or Memory Channel [32]), the costs of sending and receiving messages via the kernel is very high as well in user-level shared-memory. Thus, coherence units must be large in order to amortize the overhead of messaging. This further complicates the coherence protocol, which must support weak memory models in order to fight false sharing.

In SC-COMA, by contrast to user-level shared-memory, we integrate the software coherence handlers under the kernel level, on top of the bare hardware (Figure 3.8 (b)). This implementation makes shared-memory become completely transparent to the application level and to most of the kernel itself. In effect, SC-COMA's goal is to provide the user and system software layers with a system view that is as close to hardware shared-memory as possible. This is similar in concept to the way TLB misses are handled in software in many modern processors. Sub-kernel emulation of shared-memory was explored before in the Alewife project [12] and later in the software-only directory protocols [33], but at a more limited scale of attributions for the software layer.

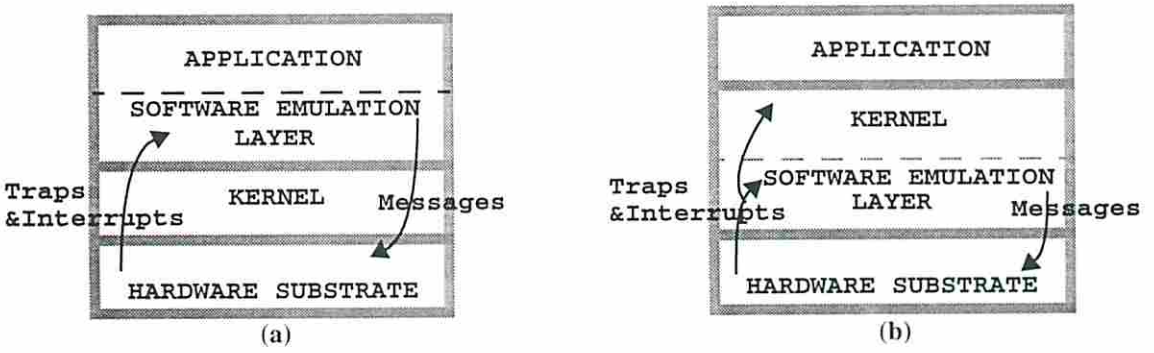


Figure 3.8. User-level versus sub-kernel emulation of shared-memory

In a user-level DSM (a) the software emulation layer must cross the kernel layer to reach the hardware substrate. By contrast, in a sub-kernel DSM (b), the software emulation layer is adjacent to the hardware.

An obvious advantage of sub-kernel handlers is that the latency and overheads to process coherence events is much reduced (the latency is of tens of instructions). For example, in the context of network protocols, experience with the *x-kernel* at the University of Arizona [60] has shown that a kernel-level protocol can outperform user-level protocols by a factor of 2 to 3. A second advantage is that the protocols are shared by all the processes running on the machine without any overhead. True, in a user-level protocol, each user could specify a different protocol optimized for each application [25]. However, this feature may be hard to exploit by ordinary programmers, which would rather experiment with ready-made protocols from a library. Similarly, in a kernel-level approach, various (possibly parameterized) protocols are coded by kernel programmers and can be selected by individual users, achieving the same goal without exposing casual users to the intricacies of protocol design and coding. A third advantage is that a user-level network interface is not necessary. Finally, the kernel itself can make use of shared memory[1].

Sub-kernel coherence handlers work with physical addresses for the coherence units, matching the addresses recorded by the ACD during a miss. Thus, no reverse address translation is required, as is the case in user-level protocols executed by dedicated processors, like Typhoon [62]. Also, this enables the coherence handlers to work in any context, not just that of the associated application. This provides extra flexibility to the process scheduling policy and its implementation.

3.3.3 Efficient Implementation of the Coherence Handlers for SPARC Processors

Because the coherence threads are very lightweight threads that do not require operating system involvement, no user-level information, except for the content of regis-

1. Some awareness is required on the part of the operating system. Shared memory should not be accessed during critical sections where exception handling is disabled. Cache-coherent DMA transfers to shared memory are not possible. Also, the virtual memory management of shared memory needs modifications.

ters, must be saved prior to their execution. The SPARC architecture, assumed by SC-COMA, provides windowed registers, which allows a very efficient implementation of the switching to and from the coherence threads. Upon an exception or interrupt, a fresh set of registers is available for the exception handler. If the handler refrains from using the common registers (floating point and global), there is no need to save all registers, thus making the thread switching a question of just a few (about 25) instructions. Even the few registers that need to be saved are stored in the trap window local registers (see Figure 3.9), thus avoiding memory references.

SC-COMA's coherence handlers are written in the high-level language C. A prologue and an epilogue, written in assembly language, arrange for the thread switching described before. For memory misses, the prologue and epilogue make it appear as if the C routine was spontaneously called just before the faulted load/store instruction. This is similar in functionality to memory-informing operations [38], but SC-COMA's C routine runs in privileged mode. The prologue and epilogue are described in more detail later and they are also listed in Appendix A.

The C routines for the MEXC and CINT handlers contain flat code (no calls to other routines). This helps in several respects. First, the chances for register window overflows due to the coherence activity are minimized, thus, reducing interference with the application. Secondly, we can keep a tight control on the stack requirements for the coherence handler, which makes it possible to use a small dedicated stack. Finally, the compiler has a large window for code optimization and can avoid storing operands in memory. In fact, all the needed operands fit in processor registers (the gcc compiler also allows us to prohibit the use of global registers, which the prologue did not save). This speeds up the handler and reduces cache pollution.

The implementation of the coherence protocol in C code has offered tremendous flexibility and ease of development. We have been able to quickly experiment with many

ideas presented later and perform many iterations over the protocol design. Portability is also enhanced, because we have used macros for system-dependent operations, such as messaging and communication with the ACD or the caches. By changing these macros, for example, we have been able to link the coherence handlers directly into a simulator of the hardware counterpart of SC-COMA.

3.3.4 Access Restart and Closing the Window of Vulnerability

Thus far, we have explained how the coherence handlers bring in the local memory a missing datum (or upgrade its state). What remains to be shown is how to complete a missing access that triggered these coherence actions. The simple solution of letting the MEXC handler return to the faulted instruction and re-executing this instruction in its application context is not acceptable for two reasons. First, potentially buffered requests require processing *after* the faulted access has been completed. Once control has been passed back to the application, after the MEXC handler terminates, there is no transparent way to invoke a routine to deal with the buffered requests from the application context[1]. Secondly, there is a potential for livelocks caused by the window of vulnerability.

As pointed out in the Alewife project [43], when a memory transaction has multiple phases, there exists a critical interval of time between the arrival of the data and the moment when the access is completed (i.e. the write is performed or the load is bound). Intervening requests for the data during this window of vulnerability, if acknowledged, could create livelock situations where none of the processors involved can make progress. In SC-COMA, the window of vulnerability opens right after the MEXC trap handler returns. CINT handlers could "sneak in" immediately after the return-from-trap instruc-

1. Actually, there is one, but too complex. It requires a special block state. When such blocks are referenced, the ACD does not fault the access. Instead it generates an asynchronous interrupt, which will be handled after the access is completed.

tion as a consequence of re-enabling traps just after leaving supervisor mode. Since there is no mechanism to indicate whether the missing instruction was performed or not, the data can be "stolen" by the CINT handler and, when the missing instruction finally executes, it will miss again, creating a potentially endless chain of requests.

SC-COMA's original solution to close the window of vulnerability consists of a combination of software locking, manipulation of the processor interrupt levels, and self-modifying code techniques. The following sequence of events takes place upon receiving a reply, without there being a chance that external coherence requests may remove the data:

1. the CINT reply handler places the data in the attraction memory and, before terminating, signals the MEXC handler that a reply was received.
2. the MEXC handler notices the reply signal and re-executes the missing instruction.
3. the MEXC handler responds to buffered requests for the data.
4. the MEXC handler returns to the application context, skipping the missing instruction.

Each of the above four actions are atomic. The interrupt level is raised so that no other network interrupt can occur. Also, no memory exceptions can occur because shared memory is not referenced in the handlers.

After issuing the request, the MEXC handler lowers the interrupt level to allow for coherence interrupts to occur in order to service external requests and to be able to receive the requested data. Requests received for the expected data are stored in a software buffer and handled in step 3. The mechanism for detecting such situations takes advantage of the fact that only one miss can be pending at a time in a node. It consists of a multi-value lock containing either the address of data for a pending miss or a special value indicating no miss is pending. Request handlers always check this lock's value against the address in the request message. Upon coincidence, the request is buffered. This cacheable lock also allows us to avoid storing a pending block state in the uncacheable tag&state table.

Instead, the MEXC handler updates the table with the final state as soon as a request has been issued, thus during otherwise idle time. This makes reply processing faster as well.

The multi-value lock is reset only in step 4, thus preventing external requests to remove the data before the access is completed in step 2 and buffered requests are answered with valid data in step 3. Once the processing of buffered requests has begun, no other requests are allowed to be buffered. This is done by raising the interrupt level to block all coherence requests. The interrupt level remains high throughout step 4 until the multi-value lock is cleared and requests cannot be buffered anymore.

Performing the missing access (step 2) is accomplished using self-modifying code. The missing instruction is copied from its location into a special slot in the code for the MEXC handler epilogue. This takes place off the critical path for miss handling, so there is no performance impact. The missing instruction slot is preceded by code that restores the CPU to the application context (except that it is in supervisor mode) and is followed by code that restores back to the MEXC handler context. On the SPARC processor this is performed very efficiently using the register window mechanism, illustrated in Figure 3.9.

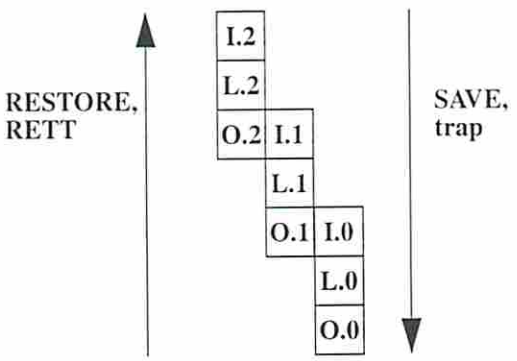


Figure 3.9. Register windows in the SPARC processor

I, L and O are the In, Local and Out registers for a window. Windows overlap and are organized in a circular stack. Window 2 is the application window, window 1 is for the MEXC handler prologue/epilogue (the trap window), and window 0 is for the MEXC C routine. The faulted instruction must be re-executed in window 2.

The user registers are already present in the previous window if miss handling is blocking (all other system activity is disabled) and all that is required is a *restore* instruction. (If, due to an overflow during non-blocking miss handling, the user registers have been spilled on the stack, they must be explicitly reloaded before issuing the restore.) The missing instruction is then executed and, with a *save*, the context of the MEXC handler is re-entered. This unusual code sequence must be performed with the traps disabled to avoid overwriting the MEXC handler registers if traps could occur while in the application window. This is also the reason why the *restore* to the application window cannot be allowed to underflow. After re-execution, the user context and the MEXC handler context share some information, such as the global registers. With careful coding of steps 3 and 4 in the handler, we avoid overwriting this common information, thus cutting the costs of saving it after step 2 and restoring it just before returning to the application. Returning is done at the address of the next instruction, pointed by the NPC register, thus skipping the missing instruction which has already been executed.

The page containing handler code and the special slot for re-executing instructions is RWX for supervisor mode and inaccessible from user mode for protection reasons. The re-execution of the missing instruction in the MEXC handler, in supervisor mode, is safe because the instruction has already passed the TLB check stage the first time it executed.

Table 3.1 reviews the list of events involved in handling the miss. ET is the Enable Traps bit in the processor status register. Interrupts are not acknowledged and traps should not occur when this bit is zero. The PIL is the processor interrupt level. Lower or same-level interrupts are ignored when the PIL is not zero. The highest PIL is 15. The SPARC processor window advances on every trap, as well as on *save* instructions, typically as part of calling a new function. The window is rolled back when *restore* instructions are issued prior to returning from a function call or trap handler. In the table we indicate the register window after the corresponding event has taken place. *Wait Reply* is a software flag used

by the CINT handler to inform the MEXC handler that a reply was received. *Pending* is the multi-value lock with the address of the pending miss. Events 2-4 constitute the first phase of the memory transaction: sending out the request. During events 5-9, the processor awaits and handles the reply. The window of vulnerability is between 9 and 11. Buffered requests for the pending exception are answered in step 13, and the application is resumed afterwards.

	Event Timeline	SPARC Window	ET	PIL	WAIT REPLY	PENDING
1	Application	n	1	0	UNLOCKED	NONE
2	MEXC prologue	n-1	0	0	UNLOCKED	NONE
3	mexc()	n-2	1	15	UNLOCKED	NONE
4	mexc() ret	n-1	1	15	LOCKED	address
5	MEXC wait	n-1	1	14	LOCKED	addr.
6	CINT prologue	n-2	0	14	LOCKED	addr.
7	get_reply()	n-3	1	15	LOCKED	addr.
8	get_reply() ret	n-2	1	15	UNLOCKED	addr.
9	CINT epilogue	n-1	0	14	UNLOCKED	addr.
10	`restore` to appl.	n	0	15	UNLOCKED	addr.
11	Instr. re-execution	n	0	15	UNLOCKED	addr.
12	`save` to MEXC	n-1	0	15	UNLOCKED	addr.
13	treat_buff_rq()	n-2	1	15	UNLOCKED	addr.
14	MEXC ret	n	0	0	UNLOCKED	NONE
15	Application	n	1	0	UNLOCKED	NONE

Table 3.1. Event list for miss handling

3.4 The Network Interface

SC-COMA assumes a loosely-integrated network interface, essentially a bus-connected device like any other I/O interfaces. Direct Memory Access (DMA) is available between the incoming/outgoing network buffers and the main memory to transfer lines without the involvement of the main processor and without polluting the caches. The net-

work interface can interrupt the main processor whenever there is a message in the incoming buffers.

3.5 Data Placement in the Attraction Memory

Techniques for data placement in the main memory of COMA architectures have received lesser attention than techniques for CC-NUMA [52]. Part of the reason is the lesser performance impact of data placement in COMA, due to automatic data migration and replication at the main memory level. This, however, does not mean there are no benefits from careful, yet application-independent, data placement in COMA. There are two aspects to be considered: the mapping of virtual pages onto physical memory, managed by the operating system, and the mapping of the physical address space onto attraction memory sets and home nodes, managed by the hardware.

3.5.1 Physical Address Space Organization

Figure 3.10a illustrates the physical address space. This space has a single dimension L, the line address. Although items can be addressed at a byte granularity, they are located and kept coherent on a line basis, which is why the term *address* will mean *line address* from now on. Figure 3.10b shows the collection of attraction memories. This space can be thought of as a three-dimensional space. For each attraction memory, along the N dimension, every set (along S) can contain up to A lines.

A complete specification of the main memory organization includes two mappings, the *home node* (HN) function and the *set hashing* (SH) function:

$$\begin{matrix} HN \\ L \rightarrow N \end{matrix} \qquad \text{(Eq. 1)}$$

$$\begin{matrix} SH \\ (L \times N) \rightarrow S \end{matrix} \qquad \text{(Eq. 2)}$$

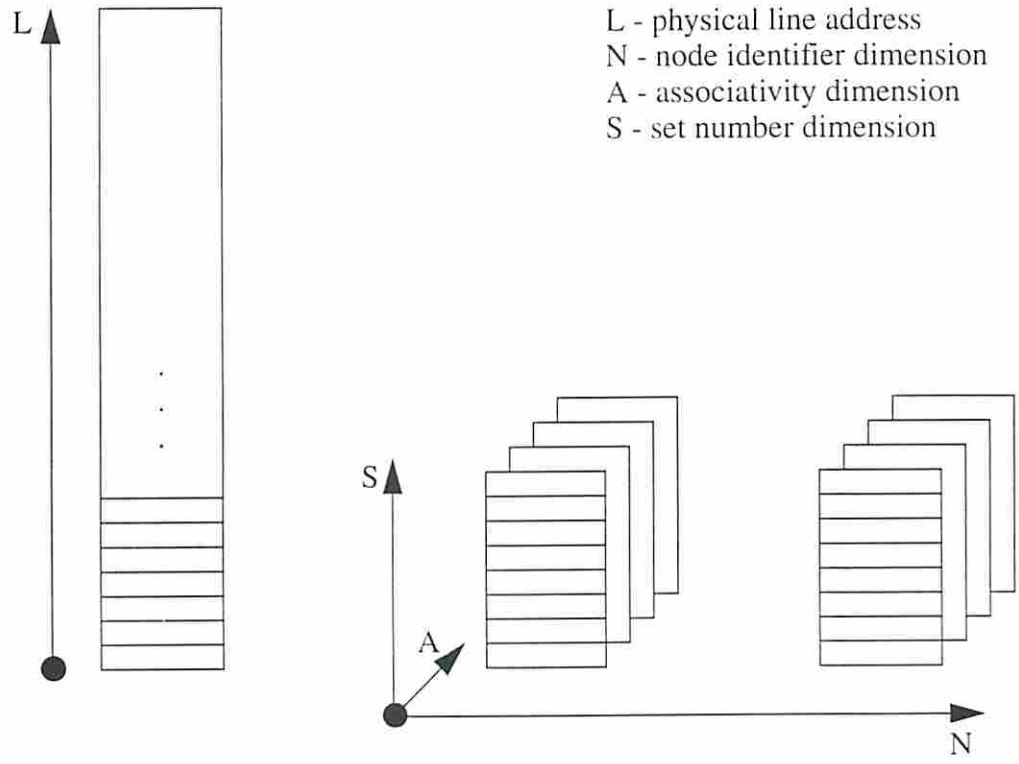


Figure 3.10. The linear physical address space and the three-dimensional attraction memories

The HN function associates a home node with each line in the physical address space. It is very common that this mapping actually operates at the granularity of the smallest page size, as set by the operating system. The reason is that centralized control over the directory information and I/O operations for swapping leads to better performance, scalability, and fault tolerance.

In the general case, as shown in Equation 2, the SH function establishes a mapping of physical addresses into attraction memory sets for each of the nodes in the system individually. Although it is very common that the hashing function is the same for every node, node-dependent hashing is sometimes used for skewed set-associative attraction memories [56]. Skewing can enhance the capacity of attraction memories to support wide replication

by disallowing any two lines to compete for the same set in more than one node. Node-independent set-hashing can create such conflicts in all the nodes, as a subset of lines are confined to the same *global set*. As will be seen, skewing is not useful for situations other than direct-mapped attraction memories, and even then, it is less efficient than other methods, such as relaxation of inclusion.

For simplicity reasons, the most common choices for the HN and SH functions are low-order or high-order interleaving. We will now examine four possibilities illustrated in Figure 3.11. The physical address is divided into fields for the offset in the line/page, home node, and set and tag. The home node field is restricted not to overlap with the page offset because the grain of allocation is set to a page.

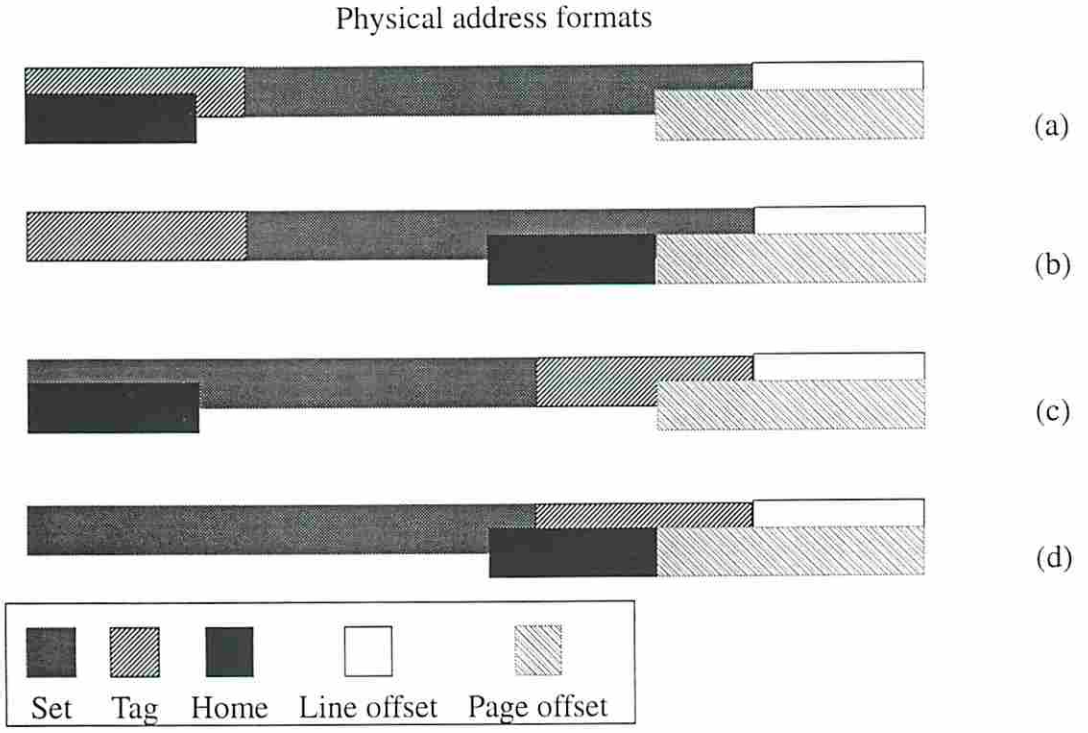


Figure 3.11. Low and high-order interleaving for the home node and set hashing functions

High-order home node interleaving and low-order set interleaving, shown in (a), is the choice for SC-COMA. This scheme avoids interference between the size of the attraction memory per node and the total size of the physical address space, thus achieving scalability. The contiguous address space per node simplifies management by the operating system, and enhances fault tolerance. Low-order set interleaving works together with the usual spatial locality properties of applications to reduce the chances of conflicts in the attraction memory, thus enhancing performance. Because of these reasons, this scheme should always be preferable over (d), low-order home node interleaving and high-order set interleaving.

Low-order interleaving for both home nodes and sets, shown in (b), offers an interesting advantage: the directory information for a global set is centralized. This creates the opportunity to optimize the replacement algorithm because the home node has information about the set's occupancy in all other nodes. It also allows requests and replacements to be combined, because they are sent to the same node. However, there are several serious disadvantages with this scheme and, as will be seen, an efficient replacement algorithm can be designed without centralized information about the global set, by using hints. The first disadvantage is that, since the home node field is part of the set field, some of the available sets will become disaffected when the number of nodes is not a power of two, thus impairing the scalability of this scheme. Secondly, low-order home node interleaving complicates the handling of super-pages (contiguous chunks of physical memory with a single entry in the page table), as they will spread over several home nodes. The scheme presented in (c) can accommodate super-pages, but may cause set conflicts due to spatial locality at the page level (the use of high and low bits for the set, with the tag in the middle, might alleviate this problem). However, scalability is just as much of a problem as it is for (b).

3.5.2 Management of the Physical Address Space

The total storage provided by the collection of attraction memories ($N \times S \times A$) must be larger than the size of the allocated physical address space by the amount allowed for replication. The allocated physical address space is managed by the operating system and it makes sense to allocate data in such a way to balance the mapping over home nodes and sets. A balanced allocation should reduce directory hot-spots at the home nodes with too much allocated data. Balanced allocation over the attraction memory sets is especially important to keep the memory pressure constant across all sets and avoid excessive conflicts and replacement traffic in overly populated sets.

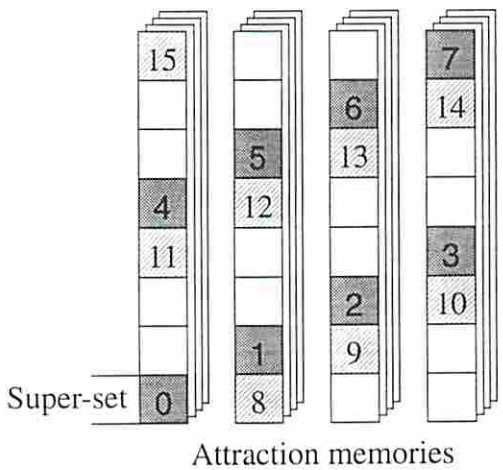


Figure 3.12. Balanced allocation of page frames in SC-COMA

Given the physical address format for SC-COMA, described in Figure 3.11 (a), a balanced allocation policy would increment both the super-set number (a super-set is the collection of LP consecutive sets, where LP is the number of lines in a page) and the home node for two consecutively allocated pages. When either the super-set number or the home node number reaches the top limit, it rolls back to zero. Graphically, this translates into the "striped" allocation shown in Figure 3.12. As pages are allocated one by one on demand,

frames are chosen along unrolling ribbons over the attraction memories, starting at set zero and ending at the top, at which point a new ribbon starts from the bottom. When the "front plane", corresponding to coordinate zero along the associativity dimension of the attraction memories, is filled up, allocation continues in the plane behind.

In the real case, the operating system must perform allocation, as well as deallocation of pages, resulting in some fragmentation of the allocated space. The overall goal of keeping a good balance over the sets and the home nodes can be achieved, in the general case, by managing stacks of free frames, one stack for each ribbon, and always allocating new entries from the shortest ribbon.

Super-pages can be easily accommodated in this scheme by allocating several contiguous frames on the same home node. Instead of always unrolling along the diagonal, the ribbons can stretch vertically (covering the super-page) and then continue their diagonal route. Clearly, the goal of balancing allocation over the sets is still met.

This allocation strategy is application-independent and somewhat equivalent to the round-robin policy devised for CC-NUMA. For situations when the application has a known optimal static allocation, which is also well balanced over the home nodes, an equivalent policy can be designed by keeping stacks of free frames associated with every node. The frames are initially put on these stacks in the order in which they are "deallocated" from the stacks associated with the ribbons.

Chapter 4

THE PERFORMANCE EVALUATION OF SC-COMA

This chapter compares the performance of SC-COMA to other systems. It begins by describing the experimental methodology. Then, SC-COMA is compared to an ideal hardware COMA in terms of execution time and speedup. Other aspects are considered. Finally, the performance of SC-COMA is compared to other NUMA memory organizations under hybrid implementations.

4.1 Experimental Methodology

4.1.1 The Simulator

We have developed a flexible simulation environment for hybrid and hardware DSM architectures. Common modules include a processor simulator, a memory management unit (MMU), two levels of caches, physical memory, a FIFO message-passing substrate, and a custom event scheduler implementing the multiprocessor features. System-specific modules interface the cache to main memory. The code implementing the coherence protocol can be linked either with the application or with the simulator by simply modifying some macros. The former case corresponds to hybrid DSM systems. In the latter case, we simulate an ideal hardware implementation where protocol handlers take zero time to execute.

The processor simulator is an efficient SPARC V8 [76] interpreter with a throughput of 1.2 MIPS on a Ultra Enterprise 3000 host equipped with a 168MHz UltraSPARC CPU. The processor simulator features trapping on exceptions raised by the simulated memory through the MEXC line. Traps are generated for over-flows and under-flows of the eight register windows as well. Also supported are leveled asynchronous processor interrupts, used for interfacing to the interconnection network. With every invocation, the simulated processor advances exactly by one instruction, as we do not simulate the details of the instruction pipeline. Load and store instructions are both blocking. The binaries of both the software coherence handlers and the application code are executed completely and faithfully by the simulator.

4.1.2 The Benchmarks

A subset of the SPLASH-2 benchmarks is compiled for SPARC V7 using *gcc-2.7.2 -O2*, and linked with the system libraries of Solaris 2.5. A special library provides routines required by the ANL macros. Support for efficient synchronization is included in the form of queue-based locks and hardware barriers. The simulator detects special load-store atomic (LDSTUB) instructions used in synchronization routines and suspends or resumes execution for threads, as appropriate. Other ANL macros, such as CREATE, WHO_AM_I, and CLOCK, along with data placement directives and all operating system stubs, employ software traps to request servicing from the simulator. The trap table and software handlers are linked together with the application to create an executable used for simulation on both the hybrid architecture and its hardware counterpart (as well as direct execution on uniprocessors). The characteristics of the benchmarks are given in Table 4.1.

Benchmark	Parameters	Total Instructions (10^6)	Shared memory (MB)
Barnes	16K particles	1,416.7	3.96
Cholesky	tk15.O	802.6	21.37
FFT	64K points	37.6	3.54
FMM	16K particles	2,373.6	29.23
LU	512 x 512	540.9	2.16
Ocean	258 x 258	302.2	15.52
Radix	1M integers	72.5	9.87
Raytrace	car	854.8	34.87

Table 4.1. Characteristics of the benchmarks

4.1.3 Data Placement Strategies and Memory Pressure Control

The default data placement strategy is round-robin with a 4KB page as the allocation unit. We point out that other placements, optimized for locality, lead to a better performance for all architectures. The improvements are very application-dependent though. It is an advantage of COMA that uninformed placement strategies perform almost as well as optimized strategies. When comparing the performance of SC-COMA to an ideal hardware implementation, a less efficient placement is unfavorable to SC-COMA, as the number of interrupts, hence the software overhead, increases due to request forwardings (three-hop transactions).

In order to show the relative importance of data placement for the performance of hybrid DSM systems, we have performed our experiments under two strategies: round-robin and best static placement. The best placement follows the comments by the authors of the code as annotated in the sources of each benchmark. Pages containing data structures with strong affinity for a processor are placed in the local memory of this processor by using run-time directives. In some cases, only a subset of the shared pages can be placed as such, and a round-robin policy is used for the rest. For *raytrace*, round-robin is the only placement scheme available.

The memory pressure is controlled by setting the size of the attraction memory (hence, the number of sets) to a value that is appropriate for each benchmark's data set size and the desired memory overhead.

4.1.4 Baseline Architecture Parameters

The simulated architectures consist of 32 nodes, each with a 200 MHz SPARC processor, a hierarchy of caches, a memory module, and a network interface. The first-level cache (FLC) is direct-mapped with 32-byte blocks. It is write-through with no allocation on write. The second-level cache (SLC) is four-way set-associative, write-back, and has 64-byte blocks. To maintain a common evaluation platform, the caches are virtually-indexed. The cache sizes are scaled down to reflect the small data set size of the benchmarks. An SLC of size between 16 and 64KB can fit the primary working set WS1 of the benchmarks [82], while at the same time yielding a reasonable ratio of memory versus cache sizes (the allocated shared memory size per node varies between 69 KB and 1.1 MB). The comparison of hardware and software COMA is quite insensitive to the cache miss ratio (just the attraction memory miss ratio matters), thus we use a 64KB SLC to speed up the simulation. A 16KB SLC is used in the evaluation of hybrid systems to induce a higher cache capacity miss ratio and better put into evidence differences between the systems. In all cases, the FLC size is set to a quarter of SLC's size.

The SLC is connected to memory by a 128-bit local bus running at 100 MHz. The memory in each node is four-way interleaved with a 16-byte wide interface and an access time of 28 cycles (140 ns). The critical word of the line is fetched first. We do not assume any fast page mode optimizations.

For COMA, the memory controller implements a four-way set-associative attraction memory with 128-byte lines. The tags and states of the lines in each set are fetched and checked in 28 cycles (one full memory cycle time) prior to the actual memory access,

as the table is stored in regular DRAM. Access checking to the code and stack segments is disabled. Tag&state table accesses by the software protocol handlers is uncached and takes 40 cycles. Intra-node read latencies are given in Table 4.2.

Read Requests	Latency (pclocks)
Hit in FLC	0
Hit in SLC	6
Hit in tag-free local memory	46
Hit in attraction memory	46+28=74
Uncacheable access	40

Table 4.2. Read latencies

The network interface is controlled through a set of memory-mapped registers. A line is transferred between the network buffers and main memory in 80 cycles, assuming DMA assistance. The simulated network is an 8-bit wide crossbar clocked at 100 MHz. The transfer of an 8-byte request takes 16 cycles and the transfer of a 128-byte line takes 272 cycles. Ten cycles are added for processing a message at the reception.

4.2 Hardware Versus Software Implementation of COMA

The experiments and results presented in this section quantify the loss of performance due to the software implementation of a COMA protocol. As the attraction memory hit ratio remains unaffected, this loss is caused primarily by the higher remote access latencies and secondly by the application processing bandwidth diverted to protocol activity. SC-COMA's remote latency add-on to the network delays is due directly to the longer processing time for protocol actions and indirectly to the larger delays spent by requests in the queues of high-occupancy protocol engines.

The hardware COMA we compare against has an aggressive controller with zero occupancy. Thus, there is no delay in the processing of a request except for contention at

hardware resources: cache, bus, and memory. Also, the processing bandwidth is dedicated entirely to applications.

4.2.1 Anatomy of a Remote Access and Micro-Benchmarking

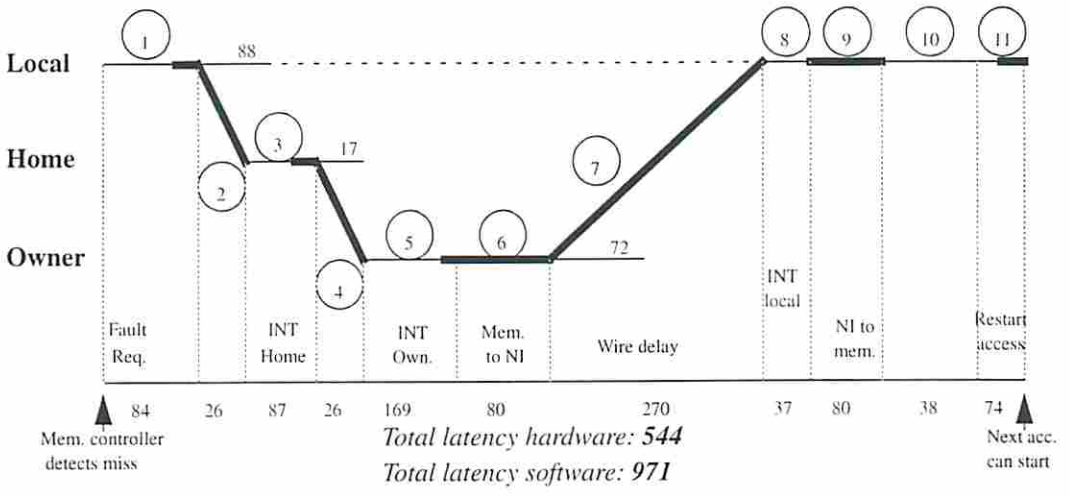


Figure 4.1. Anatomy of a remote access

The anatomy of a remote access is shown in Figure 4.1. The sequence of events and their latencies correspond to an unloaded (no contention) three-hop transaction where the local node requests to read a line and the home node forwards the request to the line's owner. Thin lines are for software latencies and thick lines are for hardware latencies in 5ns cycles. After sending the request (1) the local processor idles. The request message traverses the network to the home node (2). Upon arrival, it interrupts the home, which performs a directory lookup (3). The request is forwarded to the owner and received (4). The owner node handles the request (5). After locating the line in its physical memory, invalidating/flushing it from the caches, the processor transfers the line to the network interface (6). The line crosses the network back to the local node (7). At reception, the local node is interrupted. The reply handler (8) precedes the transfer of the line from the

NI to memory (9). After resuming the context of the waiting loop and exiting it (part of 10), the instruction is re-executed (11). The final events are checking for buffered requests, restoring the application context and returning from the fault handler; their penalty has been lumped into phase 10.

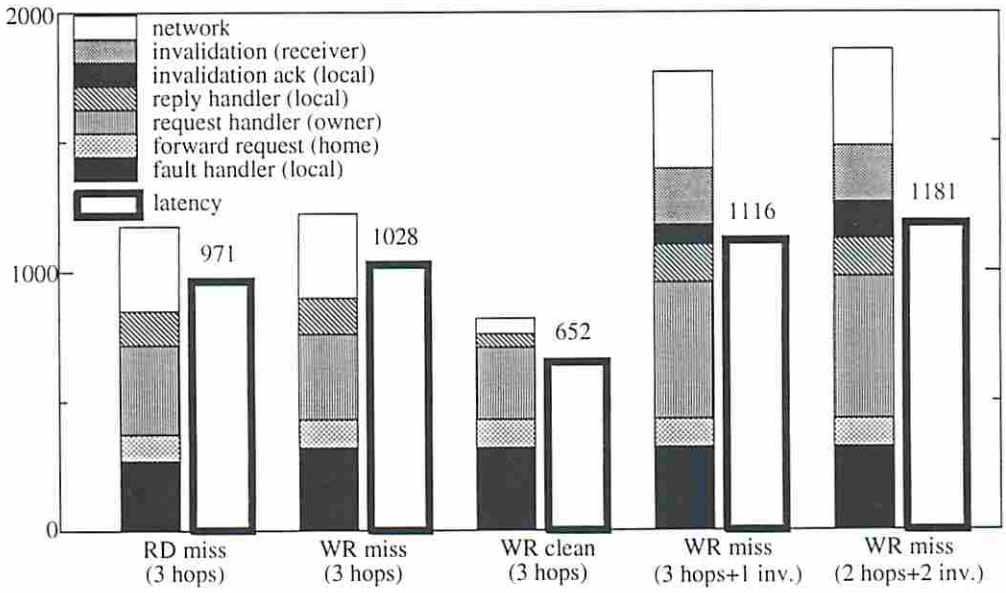


Figure 4.2. Breakdown of remote latencies for simple requests in SC-COMA

To understand where time is spent during a miss in SC-COMA, we have run a set of micro-benchmarks, each targeting a particular situation such as a read miss or a write miss with zero, one or two invalidations. For each of these cases, we measure the time spent in the different software handlers and in the network. In Figure 4.2, each stacked bar gives the time breakdown in cycles for the activities of a request. All handler times include the time to save and restore the context of the processor. The memory exception handler time also includes the time to re-execute the faulted instruction. For write misses with invalidations, the invalidation acknowledgment handler time is the average sum of all the

times spent in processing acknowledgments. The bar on the right of each stacked bar gives the latency experienced by the requester. They are shorter than the sums of the times for all activities because some activities overlap.

The most important component of the latency is the remote read or write interruption handler at the owner. The owner must identify the location of the line, using hardware assistance, and reply to the request. On average, this takes 280 cycles when no line is sent back and 330 cycles when the reply contains a line. For write requests, the first invalidation sent by the owner increases the handler time by 246 cycles (on a 32-node configuration); each additional invalidation adds 22 cycles.

On a node with a Shared copy of a line, the invalidation interruption time is 218 cycles (we only show the time for one invalidation interruption, since all the invalidations handlers are executed in parallel.) A big part of this latency is hidden, however, since the invalidation acknowledgment message is sent before searching for the line and invalidating it. Receiving an invalidation acknowledgment at the local node consumes 65 cycles, except for the last one (76 cycles).

By looking at the latency times experienced by the local node (right bars for write miss cases), we see that the global cost per invalidation is 65-88 cycles. Since the number of simultaneous copies of a line is usually low in typical programs we do not think that dispatching invalidations and collecting invalidation acknowledgments in hardware, as advocated for software-extended protocols [12], would yield a large performance improvement, except for benchmarks with wide sharing.

4.2.2 Execution Times

In this section we present the overall performance of SC-COMA for six SPLASH-2 benchmarks by comparing it to an ideal hardware counterpart, called HW-COMA. HW-COMA uses exactly the same coherence protocol as SC-COMA, but incurs no penalty for

the execution of coherence actions, as if they were performed by an extremely fast hardware controller. HW-COMA's controller is arbitrated between the local cache and the network interface and is occupied by a request for a duration of time involving mostly data transfers between memory and the network interface or the cache.

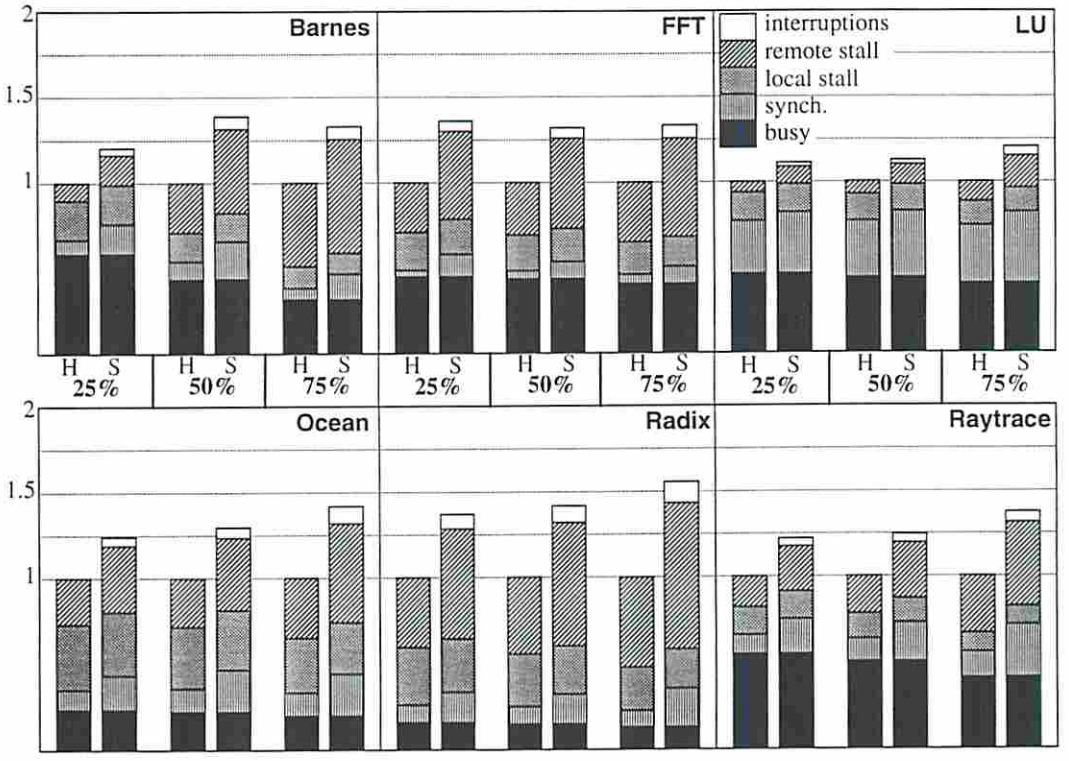


Figure 4.3. Execution times for the SPLASH-2 benchmarks
The execution time of SC-COMA is normalized with respect to HW-COMA at the same memory pressure.

Figure 4.3 shows execution times on SC-COMA normalized with respect to HW-COMA, for three memory pressure points: 25%, 50% and 75%. The execution times are broken down into several components. The *busy* time corresponds to the effective instruction processing time of the processor. The processor is stalled during *synchronization* events and whenever it misses in the FLC. When the access hits in the SLC or local memory, the delay is counted as *local* stall. In SC-COMA, this corresponds to accesses per-

formed without software intervention. Attraction memory misses contribute to the *remote* stall. An additional component of the execution time in SC-COMA is due to the processing of external requests which *interrupt* the application thread. This category excludes the overhead of requests occurring while the processor spins at a synchronization point or while a remote access is pending. SC-COMA's slowdown ranges from 11-37% at low memory pressure to 21-56% at high pressure. In spite of almost identical cache and memory hit ratios for the two architectures, the local stall is higher in HW-COMA. This is explained by the increased delay in processing cache misses, when HW-COMA's controller is busy with external requests. By contrast, SC-COMA's cache controller has exclusive access to the bus and the memory.

The amount of remote stall in SC-COMA, as compared to HW-COMA, roughly scales up by the ratio of the remote read/write miss latencies, shown in Figure 4.1. The reason is that the node miss ratio remains practically constant in the two architectures and, except for LU, there is no significant component of upgrade (ownership) misses. Compared to read misses, the overheads of SC-COMA for upgrade misses are relatively bigger, hence LU shows a higher scale-up factor for the remote latency. Other factors of fluctuation from this approximate ratio are the amount of request forwarding and contention for certain nodes, which increases queuing delays. The number of forwarded requests should decrease with better placement strategies, leading to a reduction of the average latency which is more significant in SC-COMA. The likelihood of contention goes up with the memory pressure, as processors become interrupted more frequently. This explains why the ratio between the amounts of remote stalls increases slightly with the memory pressure.

The activity of the coherence handlers affects the synchronization stall indirectly. The increased synchronization penalty in the context of software-implemented protocols has been attributed by Grahn and Stenström [33] to node activity imbalances due to

uneven distributions of coherence requests. This is more serious at high memory pressure, when the protocol overhead is more pronounced.

Finally, SC-COMA has a component of overhead due to interrupts disturbing the application. Overall, this is quite small, indicating that a potential communication coprocessor would be under-utilized. As memory pressure increases and replacements become more frequent, this component becomes more significant, but never critical. An interesting effect in some applications is the occurrence of external requests when the processor is stalled anyway, either in synchronization or because of a pending miss. LU, with a high synchronization penalty, is able to overlap the processing of some external requests with barrier synchronization. On the other hand, FFT, Radix and Ocean exhibit clustered misses during data exchange phases, when processors are cross-servicing misses, and some of the overhead due to interrupts is again absorbed in the remote stall.

4.2.3 Effects of Protocol Engine Occupancy

The access latency presented in Figure 4.1 was in the absence of any contention. For the real workloads analyzed previously, the average remote latencies are different because messages do have to wait in buffers until the protocol engine (i.e. the processor) picks them up for processing. The queuing delays for messages usually become higher as the servicing time increases.

Table 4.3 shows the average queueing time per message in SC-COMA when the memory pressure is 75%. There are several messages involved in the processing of a remote miss, thus the total contribution of queueing to the remote latency is higher. The other data in the table indicates the average read and write remote latencies. Although there is a mix of two and three-hop remote accesses, the latencies are clearly influenced by contention for the protocol engine, as indicated by the message queueing time.

Benchmark	Average Queueing Time (cycles)	Average Read Latency (cycles)	Average Write Latency (cycles)
Barnes	54.07	1083	820
FFT	89.21	1212	942
LU	57.24	1038	759
Ocean	98.92	1323	774
Radix	124.57	1266	1286
Raytrace	43.45	923	939

Table 4.3. Average message queueing time and latencies

Contention at the protocol engine is in direct proportion to the average node miss ratio. Radix, at one extreme, and LU, at the other, are representative of this effect. Contention is also higher in applications with bursts of misses during regular communication phases. The matrix transposition phase in FFT and the all-to-all communication phase in Radix are examples of this effect. On the other hand, irregular communication, exhibited by Barnes and Raytrace, does not generate temporal hotspots of coherence activity.

4.2.4 Speedups

In Figure 4.4 we present speedups for up to 32 processors. The algorithmic speedup is derived for a perfect memory system (no stalls). To gain insight into the behavior of the COMAs, we show the speedup at three different memory pressures. For a given application and memory pressure, regardless of the number of processors, the total amount of attraction memory in the system is kept constant. Also, we do not scale the processor cache size with the number of processors. The tag-checking overhead in shared memory accesses is removed for simulations of the uniprocessor case. This explains why the slope of the speedup is smaller than one right from the start (i.e. for just a few processors), especially in applications with high cache miss ratios: Radix, Ocean, and FFT.

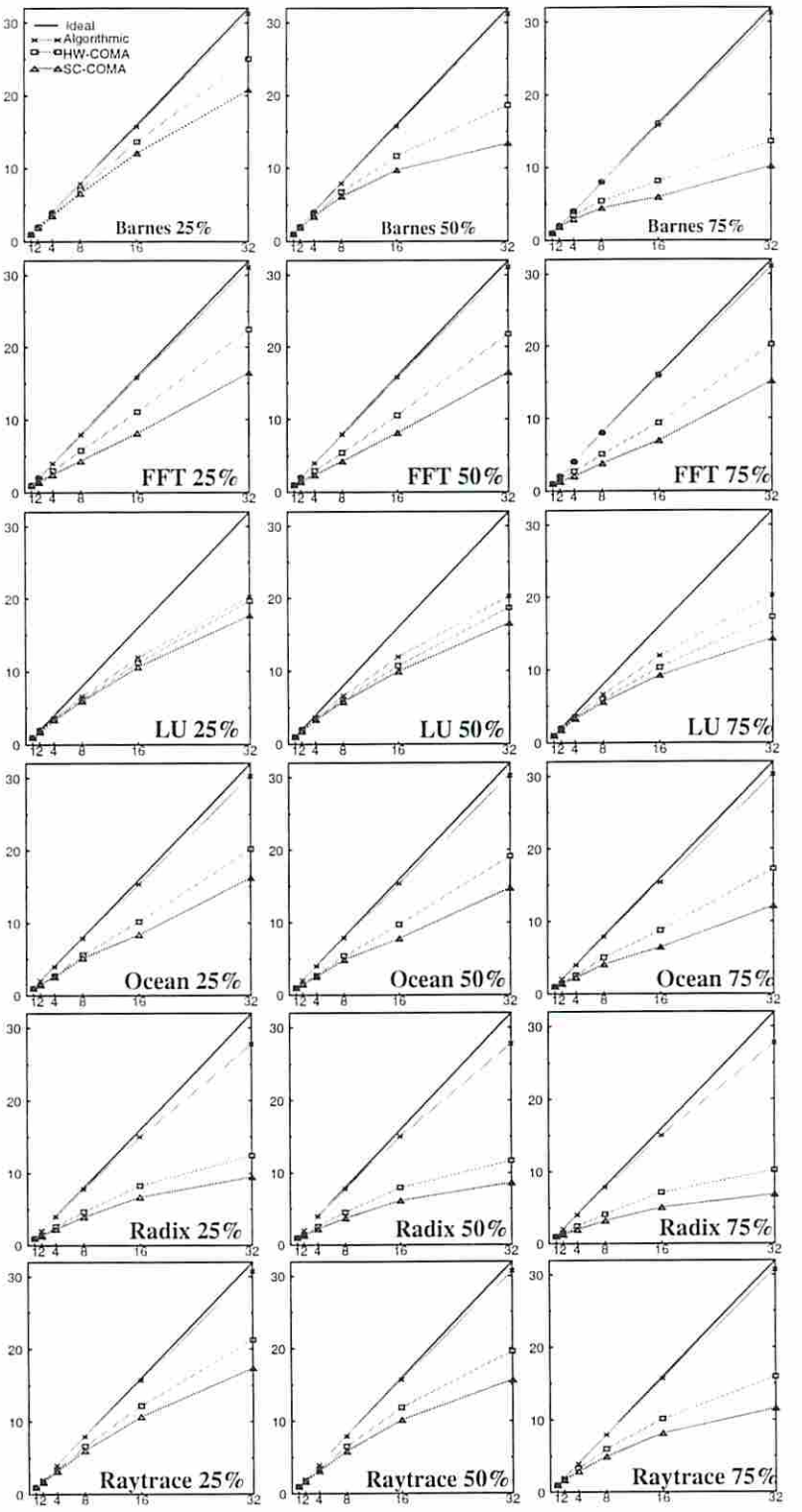


Figure 4.4. Speedups for up to 32 processors (three memory pressure points)

The main observation is that, while SC-COMA does show some slowdown, it does not introduce any bottlenecks accentuating the speedup saturation. The communication-to-computation characteristic of an application dictates when saturation occurs. The higher remote latency in SC-COMA makes it's speedup diverge from HW-COMA's, but the speedup still remains almost linear when memory pressure is low. Radix is the only application with a traffic so large to show rapid saturation for both COMAs. At higher memory pressures, the replacements and the capacity traffic start to pick up, deteriorating the speedups. Capacity traffic increases mostly in Barnes and Raytrace. In Raytrace, where data is mostly-read, the rate of replacements is small and the combined effect on the speedup is lesser.

Let's introduce a simple model to further discuss the speedup. The execution times for the hardware and software systems using P processors are given by:

$$T_{HW} = \frac{B + RL}{P} \qquad T_{SC} = \frac{B + R(L + \Delta L)}{P} \qquad \text{(Eq. 1)}$$

B is the total cycles of computation and local memory stalls, R is the total amount of remote references, L is the communication latency, and ΔL is the increase in latency due to the software implementation. The software slowdown is then given by:

$$\frac{T_{SC}}{T_{HW}} = 1 + \frac{\Delta L}{L + B/R} \qquad \text{(Eq. 2)}$$

In Figure 4.5 we have plotted in solid lines the slowdown given by Eq. 2 as a function of the latency L for different ratios B/R and assuming $\Delta L=500$. The dotted lines represent the predicted slowdown for the six benchmarks, by using the B/R factors obtained through simulations of 32-node systems for a 75% memory pressure. The vertical dashed line corresponds to the latency in the hardware system.

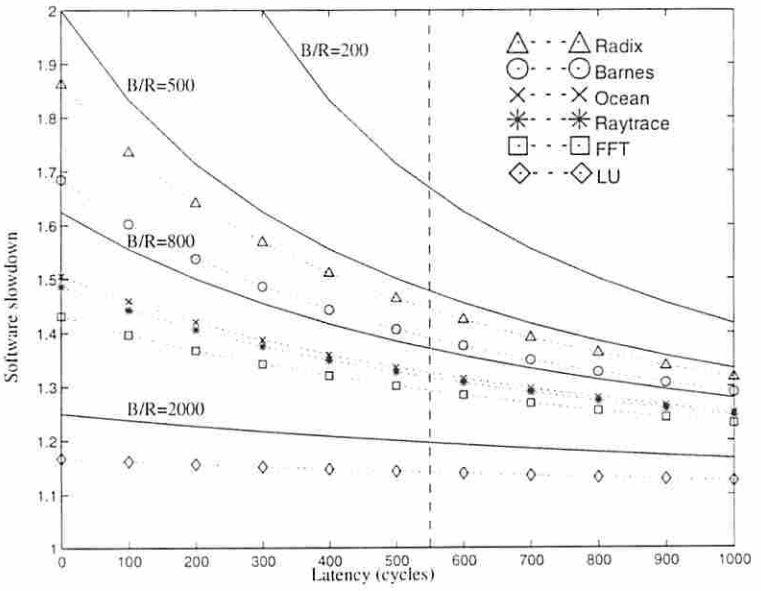


Figure 4.5. SC-COMA slowdown from the analytical model ($\Delta L=500$, $P=32$)

As expected, the ratio *B/R*, which is strongly related to the computation-to-communication ratio of the applications, has a significant impact on the software slowdown and on the speedup of both systems. LU, with the highest *B/R*, shows minimal differences between the HW and SC speedups. On the other hand, Radix, with the lowest *B/R*, exhibits a serious degradation of speedup on both platforms. The other applications also show software speedup degradations which can be ranked in the order illustrated by Figure 4.5.

A second observation from Figure 4.5 is that the vertical line intercepts the slowdown curves in a relatively flat area, after their knee, when $B/R>500$. This is a very likely case for many scientific and image processing applications, including the six SPLASH-2 benchmarks. Thus, a low-overhead software implementation will predictably perform well (10-50% slower) across a wide range of communication latencies that are typical for a NOW (550 cycles is still aggressive). This also holds true for a wide range of *B/R* ratios.

As such, it is reasonable to expect that the software speedups will not show signs of saturation that are not already present in the hardware speedups, unless *B/R* drops significantly below 500

The presented model has several shortcomings. Synchronization effects are ignored; synchronization penalties tend to be higher under software coherence. ΔL, assumed to be constant in Eq. 2, varies across applications and depends on the number of processors because of queuing delays, message forwardings, and the amount of ownership misses. Similarly, the average hardware latency can vary around 550, but we have seen lesser sensitivity to this. Finally, the evolution of *B/R*, as the number of processors changes, shows sudden jumps due to caching and data placement effects, and, sometimes, to changes in the communication patterns. Nevertheless, Table 4.4 shows that the actual and the modeled software slowdowns are very close.

	Barnes	**FFT**	**LU**	**Ocean**	**Radix**	**Raytrace**
B/R	730	1160	2990	990	580	1030
Slowdown (model)	1.39	1.29	1.14	1.33	1.44	1.32
Slowdown (actual)	1.33	1.34	1.20	1.42	1.49	1.38

Table 4.4. Actual and modeled software slowdowns

4.2.5 Effects of the Processor Speed

It is expected that, with increasing processor speeds, the overhead of coherence-related software and the contribution of software-implemented actions to the remote latency should be relatively diminished, if the memory and network speeds are kept constant. In order to quantify this intuition, we have performed simulations for SC-COMA and HW-COMA using varying processor clock frequencies, up to 1GHz (the horizontal axis in

Figure 4.6 can also be read in term of MIPS, thus capturing effects in multiple-issue processors). Our indicator is SC-COMA's slowdown, the ratio of execution times T_{SC}/T_{HW}.

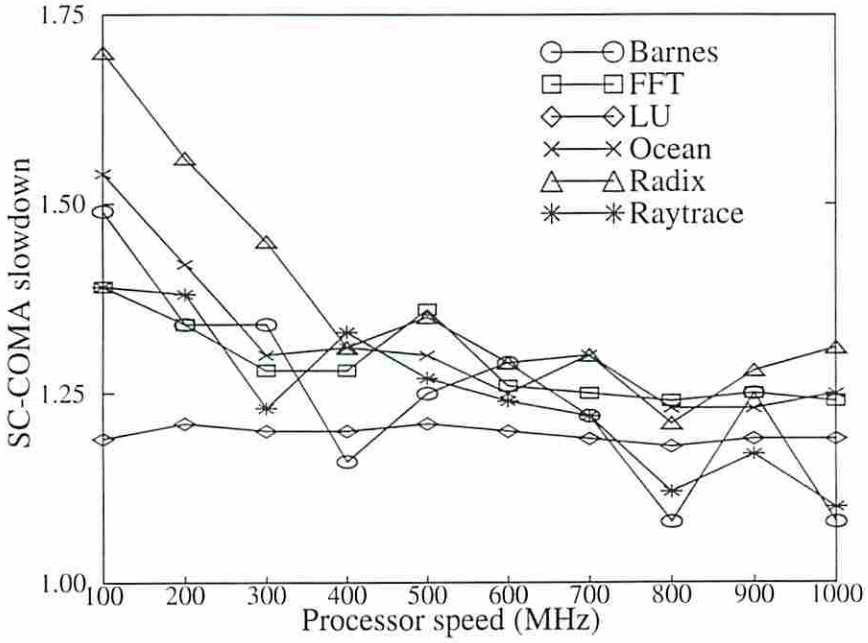


Figure 4.6. SC-COMA's slowdown for different processor speeds

Barnes	FFT	LU	Ocean	Radix	Raytrace
1.08	1.24	1.19	1.25	1.31	1.10

Table 4.5. Asymptotic slowdown for SC-COMA (estimated at 1GHz)

Figure 4.6 shows a clear improvement for SC-COMA as the processor is clocked faster and the overhead of the software-implemented protocol is reduced by comparison. The inconsistent behavior in Barnes and Raytrace may be due to dynamic work scheduling, which could lead to different execution paths. LU does not show any improvement, possibly because of the high synchronization penalty and the small amount of remote stall. Finally, SC-COMA's slowdown never approaches one. This is because HW-COMA has instantaneous access to all the coherence tables and the directory, whereas SC-COMA

must perform costly uncached accesses. Table 4.5 lists the estimated value to which SC-COMA's slowdowns converge. Figure 4.6 shows that the convergence is rapid.

To explain and better understand the observation captured in the plots of Figure 4.6, we now introduce a simple model. The execution times on the two platforms are:

$$T_{HW} = Ct_C + Mt_M + Rt_R \qquad \text{(Eq. 3)}$$

$$T_{SC} = Ct_C + Mt_M + R(t_R + S_I t_C + S_M t_M) \qquad \text{(Eq. 4)}$$

In the above expressions, C stands for the total count of ALU cycles and cache cycles for the memory references hitting in the cache. M and R stand for the number of memory references hitting in the attraction memory or accessing remote nodes. t_C, t_M, and t_R are the processor cycle time, memory access time, and remote access time. For software coherence, a remote access incurs an extra delay due to the execution of S_I instructions and S_M memory accesses in the software handlers. S_I and S_M are appropriately adjusted to account for the overhead of interruptions, off the application's execution path. S_M accounts only for the uncacheable accesses to the Tag Table and hardware registers and we assume there are no cache misses during the execution of the software coherence handlers. Then the slowdown s is given by

$$s = \frac{T_{SW}}{T_{HW}} = \frac{(C + S_I R)t_C + Mt_M + Rt_R + S_M R t_M}{Ct_C + Mt_M + Rt_R} \qquad \text{(Eq. 5)}$$

If t_C is the variable, the slowdown in Eq. 5 has the form of function $f(x)=(ax+b)/(cx+d)$. With positive coefficients, this function is monotonic. It is increasing if $ad-bc > 0$, and decreasing otherwise. In the particular case of Eq. 5, the monotonic increase condition is

$$C < \frac{S_I}{S_M}\left(M + R\frac{t_R}{t_M}\right) \qquad \text{(Eq. 6)}$$

For realistic system parameters, M is of the same order of magnitude as R. Since the ratio t_R/t_M in a NOW is usually larger than 5, one can approximate Eq. 6 by

$$\frac{R}{C} > \frac{S_M t_M}{S_I t_R} \qquad \text{(Eq. 7)}$$

For our particular implementation, $S_M t_M / t_R \cong 0.25$, and $S_I \cong 300$. Thus, Eq. 7 becomes

$$\frac{R}{C} > \frac{1}{4S_I} = \frac{1}{1200} \qquad \text{(Eq. 8)}$$

This means that a monotonic increase in the slowdown of the software-coherent system as a function of the cycle time t_C can be expected when more than one in about 1200 instructions generates a remote reference. Equivalently, if this is the case, the slowdown decreases as the processor frequency increases. If condition Eq. 8 is not met, as in the case of LU, one can actually see a degradation of the slowdown as the processor becomes faster, a non-intuitive result. To avoid this for a given *R/C*, the only reasonable way, as indicated by Eq. 7, is to appropriately reduce S_M, the number of uncacheable accesses on the critical path. Lowering S_I alone actually shrinks the set of applications satisfying Eq. 7. The *R/C* factors for our six benchmarks are approximately: Barnes-*1/530*, FFT-*1/810*, LU-*1/2200*, Ocean-*1/400*, Radix-*1/220*, Raytrace-*1/820*.

Eq. 5 also gives us the value to which the slowdown curves converge as the processors get faster and t_C approaches *0*. This value is

$$s_\infty = \frac{Mt_M + Rt_R + S_M R t_M}{Mt_M + Rt_R} \cong 1 + \frac{S_M t_M}{t_R} \qquad \text{(Eq. 9)}$$

This result clearly shows the impact of uncached accesses on the software slowdown. In our implementation the value of s_∞ is equal to *1.25*, which agrees with the experimental results from Figure 4.6.

4.2.6 Effects of Software Coherence on the Time-to-Market

To better understand the prospects of SC-COMA in a world where processors are clocked even faster, we have plotted in Figure 4.7 the absolute execution times for some benchmarks using processors clocked up to 1GHz. The results for the other benchmarks are similar to Barnes.

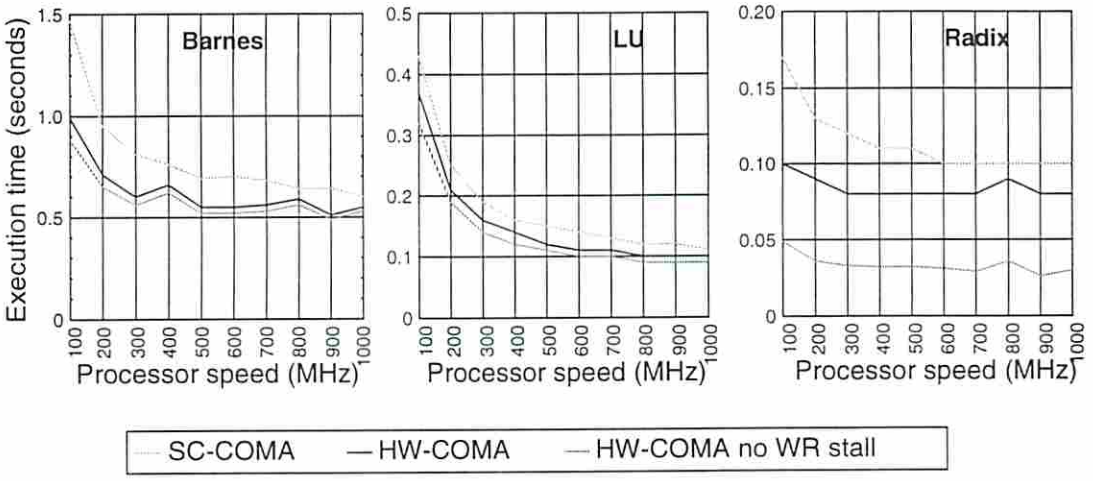


Figure 4.7. Absolute execution times at varying processor speeds

The curves for LU show that a 275 MHz SC-COMA performs like a 200 MHz HW-COMA. The same is true for a 450 MHz SC-COMA and a 300 MHz HW-COMA. In essence, this indicates that SC-COMA could be a very viable solution for the present and near-future. The release of a HW-COMA using 300 MHz processors could take as long as the development of a next-generation, 450 MHz processor, which can be used immediately by SC-COMA, at virtually no costs. On the other hand, Radix has a low computation-to-communication ratio and the execution time saturates quickly, offering less opportunity to exploit the time-to-market advantage of software coherence.

For all applications, at higher processor speeds, this advantage disappears, as the execution time starts to saturate. However, at saturation, the performance of SC-COMA is

relatively close (within 25%) to HW-COMA's, regardless of application specifics. This confirms the expectation that an ultimate performance limitation for software coherent systems is the efficiency of data movement and the overhead to control this movement.

The current version of SC-COMA is designed for a sequentially consistent hardware. Thus, the store buffers are disabled. In Figure 4.7, a third curve is plotted for a HW-COMA where all write stalls have been eliminated. At 200 MHz, SC-COMA shows a slowdown between 1.45 for LU and 3.25 for Radix, as compared to this HW-COMA with ideal release consistency. However, the software protocol in SC-COMA could be upgraded as well to run on a release consistent substrate assuming the software ability to recover pending stores from the buffer.

Current and future processors will incorporate features to fight the memory wall [66], such as bigger on-chip caches, simultaneous multithreading [78], and out-of-order execution. At the same time, limited improvements in memory speed are also expected. The net effect is that the saturation of the execution time will be pushed toward higher processor speeds. The question, then, is how do these processor features affect the performance of software coherence. Will it be able to avoid saturation just as well? Systems with external protocol processors could probably deal with some of the advanced processor features, most notably non-blocking loads, with less overhead. On the other hand, processor/memory integration trends [66][84] and memory feedback mechanisms [38] favor at least a physical collocation of the protocol engine and main processor. We believe that, in integrated systems, the memory access checking for loads/stores can be efficiently incorporated in the processing pipeline and low-overhead traps, similar to memory-informing operations [38], could start the appropriate handlers on the main processor.

4.2.7 The Performance Impact of Attraction Memory Associativity

The associativity degree of the attraction memory has a double impact on COMA's overall performance. Firstly, the attraction memory hit ratio generally improves as the associativity is increased, resulting in fewer accesses requiring software intervention. As a side effect, the number of replacements is also reduced when associativity is increased, again reducing software overhead and network traffic. Secondly, for a direct-mapped attraction memory, it is possible to eliminate the tag-checking overhead, which speeds up all memory accesses. Figure 4.8 illustrates the impact of the associativity on the execution time at different memory pressure points. The choice of a four-way set-associative attraction memory becomes quite evident. Eight-way set-associativity brings minor improvements at high memory pressure only. At 25% memory pressure, even an associativity of two seems satisfying, but as the pressure increases, especially for Barnes and Ocean, the performance degrades considerably.

The bad performance of the direct-mapped attraction memory, in spite of having no tag-checking overhead, is due to two reasons. Firstly, increased chances for conflicts translate into higher node miss ratios. More importantly, within the confinement of a global set, there is limited capability for replication. A single line L_1 cannot be efficiently shared by many nodes, especially at a high pressure, because line L_2, replaced when L_1 is replicated, will likely conflict again with L_1 in the node where the replacement request is sent. Skewing [70] alleviates this problem by using a different hashing function for each node in order to translate a line address into a set index. Thus, two lines contending for the same set in node N_1, will likely be free of contention in any other node. Thus, it is possible to efficiently replicate a line to all the nodes, without creating a cascade of replacements. We must point out that this does not come for free in a hybrid system. The software handlers themselves must compute the set when they access tables indexed by the set number.

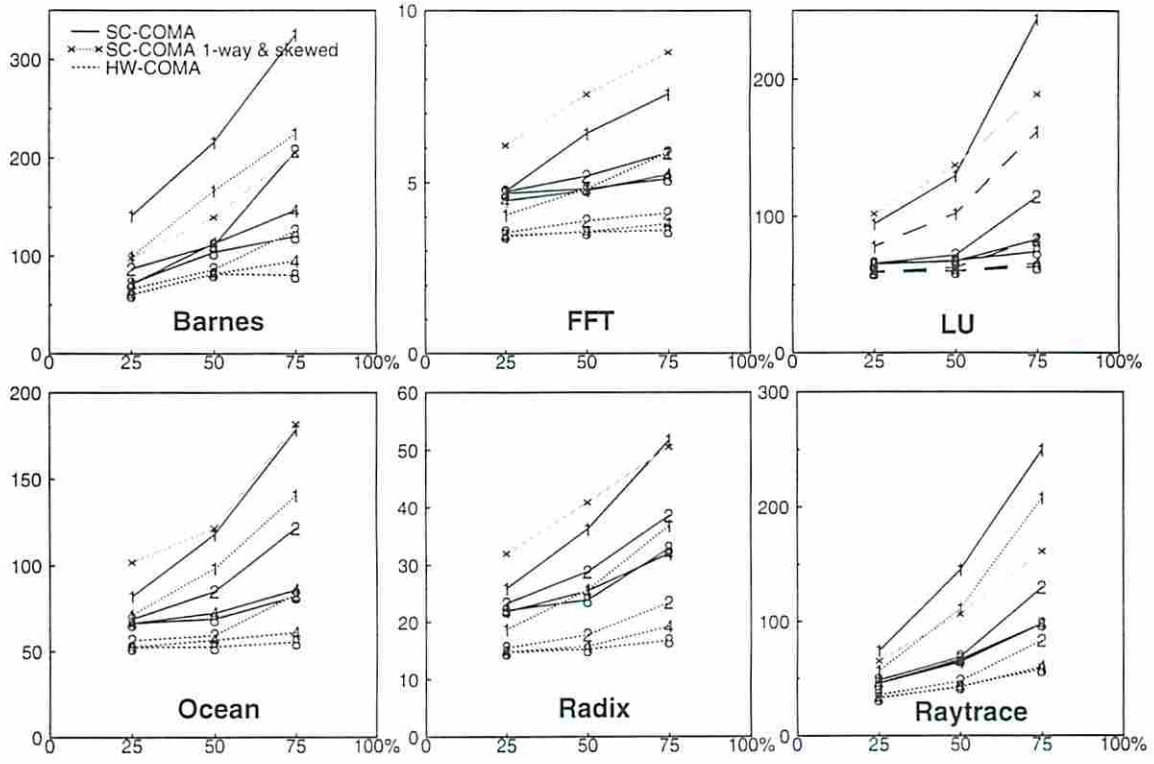


Figure 4.8. Execution times for different attraction memory associativities

It is true that hardware could assist to make skewing totally transparent, but this violates the goal of simple hardware. With dashed line, in Figure 4.8, we present the performance of a skewing scheme with four different hashing functions. The hashing functions are described by:

$$set(A,N) = ((A_{19}..A_{0})\ xor\ (A_{19}..A_{15} << (N\%4)))\ \ \%\ \ NUM\_SETS \qquad \text{(Eq. 10)}$$

where $A_{19}..A_0$ is the physical line address and N is the node number. In the shared address space, the field $A_{19}..A_{15}$ specifies the home node. As can be seen from the plots, the results are mixed. The overhead of more complex computations for the set address in the software handlers is not offset by a reduction in the number of replacements in FFT, Ocean and Radix. On the other hand, Barnes and Raytrace show overall improvements, whereas in LU skewing works better at high pressure only. It is unclear whether other skewing

schemes would bring significant changes. It seems that, with or without skewing, a direct-mapped attraction memory is not an attractive option. Furthermore, as will be seen, relaxing the inclusion property (cached blocks must have an allocated frame in the attraction memory) is a much more efficient and easy to implement method of dealing with the ineffectiveness of the attraction memory for wide replication.

4.3 The Impact of Memory Organization in Hybrid DSM

This section explores four memory organizations for hybrid DSM systems. These hybrid architectures are inspired from four hardware DSMs: CC-NUMA, RC-NUMA (NUMA with remote data cache) [85], S-COMA (Simple COMA) [67], and COMA [40]. In CC-NUMA each memory block is anchored to a home node and cannot be replicated outside the processor caches. This restriction is alleviated in RC-NUMA by a dedicated remote data cache (RDC) organized in main memory. In COMA, each memory is managed as a set-associative cache; memory blocks are associated with a home node, but may replicate freely, as in a multi-cache system. S-COMA is similar to COMA except that memory is allocated in units of pages during replication so that the mapping of blocks into memory is done automatically by the MMU. A replicated page frame is allocated empty and valid memory blocks are brought in from remote on demand. Thus the memory of S-COMA is managed as a sector-mapped cache with a sector equal to a page.

There are many performance trade-offs in these four architectures involving mostly the behavior of the memory hierarchy. To quantify and compare these trade-offs, we have completely implemented the coherence protocol handlers for each architecture, and the handlers are faithfully simulated on our execution-driven simulation platform. We evaluate six complete SPLASH-2 benchmarks [82], which were optimized for high performance hardware DSM systems (CC-NUMA) and exhibit a combination of fine-grain and coarse-grain sharing.

In the rest of this section, we will first overview the hardware and software support needed by the four architectures and follow with a discussion of the performance trade-offs. Finally, we present and discuss the results of the evaluation.

4.3.1 Hardware Support

The basic hardware substrate is a message-passing multiprocessor. Each node is made up of a commodity processor and its associated caches. The caches connect to main memory through a multiprocessor local bus. The network interface with one incoming and one outgoing memory-mapped message buffers resides on the bus. The processor may send messages by writing into the outbound buffer and the network interface generates an interrupt to the processor as soon as it receives a message. In essence, this is the standard network of workstations environment, but this communication support is at least present in any parallel computer. Additionally, a custom hardware, the Access Checking Device (ACD), extends the functionality of the memory controller to implement the fine-grained shared memory in system-specific ways, as described next.

The software protocol handlers execute on the main processor. Hence, we do not rely on the presence of a dedicated protocol processor in the node. We assume the processor features restartable load/store instructions. This is commonly available in modern processors in conjunction with MMU operations. However, given the implementation of the access control mechanisms, accesses may still trap after passing the MMU check. In this study we assume blocking loads and stores, so that memory access faults are synchronous. We assume there are software means to downgrade a cache block. Either special instructions are available for this purpose (such as *Invalidate* and *Flush*) or the appropriate bus transactions are issued by the ACD under software control.

4.3.1.1 Memory Access Control

In a hybrid DSM, every cache request to the shared memory must be validated for completion without software intervention. This process, known as *access checking* or *lookup* [71], employs two mechanisms. The *presence detection* mechanism ascertains the presence of the line in local memory, and, if a local copy is found, it also fetches the state of that local copy. The *permission checking* mechanism then checks that the state (e.g., Invalid, Shared or Exclusive) is compatible with the request type (e.g., read or write). In order to reduce false sharing effects, at least the resolution of the permission checking mechanism must be fine-grain. Accesses that fail either the presence or permission tests require software intervention.

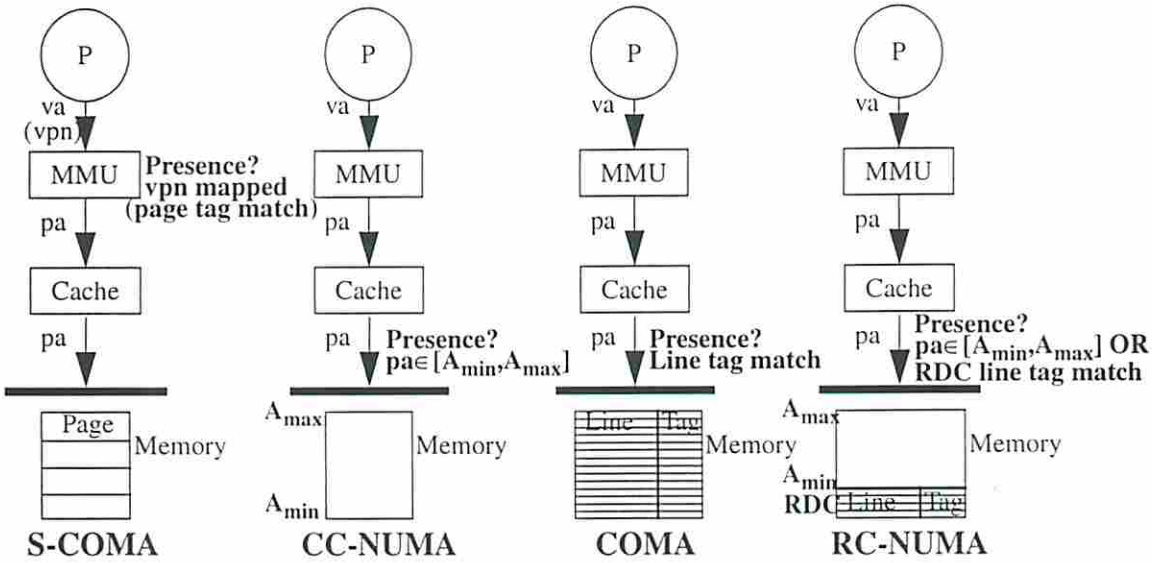


Figure 4.9. Implementation of the presence test in each architecture

In the four architectures explored in this study, the permission checking mechanism is identical, and based on a table of state bits indexed by the physical address of the memory line. However, the presence detection mechanisms are very different, as illustrated in Figure 4.9. S-COMA uses the standard virtual memory support to implement the

presence checking mechanism, at no hardware cost or performance overhead. If a virtual page has an allocated page frame, the datum is present locally at the translated address and the permission check is performed at the memory. Otherwise, a page fault signals the failure of the presence test. Thus, S-COMA resolves the presence test at page granularity and on every access. Because the sharing space is virtual and physical addresses have only local significance, the coherence protocol must use global identifiers. In the event of an access fault detected by the permission checking mechanism, the physical address must be converted to a global identifier using a reverse translation table.

In the other three architectures the presence test is performed with the physical address on cache misses only. The sharing space is physical and the coherence protocol uses physical identifiers. A partition of the shared space is assigned statically to each node in CC-NUMA. A simple comparison of the physical address with the range of local addresses answers the memory presence test. In COMA the main memory has a set-associative organization and a set of tags identifies the lines in the set. RC-NUMA, additional to the tag-free local memory, has a main memory cache which can replicate remote data, thus being a combination of CC-NUMA and COMA.

Figure 4.10 revisits the partitioning of the main storage in each node. The Tag and State Tables are consulted by the ACD on every access to check local presence and permission and are modified by the software handlers. The Directory contains the higher-level protocol information and is accessed by the software handlers only. The application memory is divided into private and shared memory. Whereas the State Table and Directory are present in all three machines, the Tag Table is only needed by COMA and RC-NUMA. In COMA, the partition of local memory hosting shared data, often called *attraction memory* (AM), is managed as a set-associative memory. It is simpler -- although not necessary -- to combine the Tag and State Tables. Each entry of this combined table packs the tags and the state bits for one entire set. Access checking is performed by fetching one table

entry, using the least significant bits of the line address as the set number, and by comparing the tags in the set with the tag of the processor address as well as checking the state bits at the same time. For CC-NUMA and S-COMA, we assume the State Table is hosted in a separate memory module, accessed in parallel with the main memory, whereas COMA's Tag and State table is stored in main memory and is accessed prior to fetching a block. RC-NUMA has both a State Table for the local memory, like CC-NUMA, and a Tag and State table, like COMA, for the RDC. In all cases, the Directory is stored in main memory.

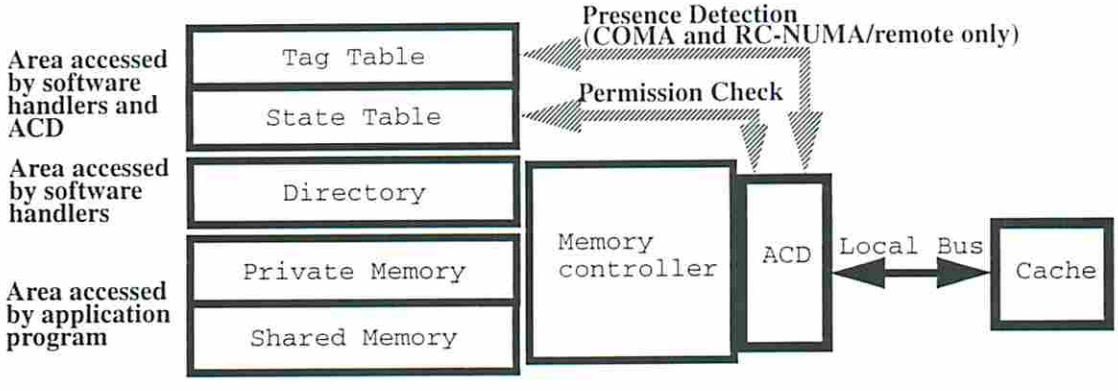


Figure 4.10. Logical memory partitioning and the Access Checking Device (revisited)

Concretely, the ACD is either incorporated in the memory controller or in a snooping device attached to the memory bus. It performs the following services on every memory access:

1. (skipped for S-COMA) decide if an access must be checked (shared) or not (private), based on the address. If the access is private, skip to 5.
2. (CC-NUMA and RC-NUMA only) decide if the bus address points to data present in local memory.

3. locate the line and its table entry.
4. fetch the state of the line and validate the access (an associative match is done in COMA and RC-NUMA/remote)
5. perform the memory access (in parallel with 4, for CC-NUMA, S-COMA, and RC-NUMA/local)
6. respond with data to the bus transaction OR fault the access using a bus error signal.

In support to the protocol handlers, the ACD also provides:

• memory-mapped registers containing information about the faulted access (address, type, etc.) and control registers.

• assistance for cache block downgrading by issuing bus transactions under software control (optional).

4.3.1.2 Special Hardware Support in CC-NUMA

By contrast with the other systems, inclusion is not maintained between cache and local memory in CC-NUMA. One important aspect of inclusion is that the write-back of victimized lines and the cache miss occurring at the restart of an instruction after an access fault are always completed locally in hardware. Three problems must be solved. First, cache write-backs need special handling when the victim line is non-local. It is unclear whether software recovery of a faulted write-back is feasible, because of the possibility of generating more write-backs during the handling. In any case, it would complicate the handlers, by forcing them to deal with recurrence. Rather, special hardware captures remote cache write-backs and transforms them into network packets sent to the home node.

Second, since non-local lines are stored only in the cache, the access fault handlers must be able to inject an incoming line from the network buffers into the cache. Instead of manipulating the cache and its tag storage, we have solved the problem with a special one-

line tagged buffer in the ACD. The line is served to the cache when the access is restarted, as if it were coming from local memory and the tag is cleared in the ACD buffer.

Finally, special care must be taken with write on read-only copies of remote lines, as the line may be displaced from the cache while the protocol handlers are executing. To solve this, we systematically request a new copy when we upgrade a remote line from read-only to read-write in the cache.

Basic and impossible-to-fix hardware shortcomings of CC-NUMA stem from the fact that the local memory cannot host remote lines. As a result, the coherence unit between nodes must be the cache line because enforcing coherence on larger units would only create false sharing. This is a pity, since, in some cases, the prefetching effect associated with a larger fetch unit in memory would benefit performance. By the same token, line prefetching in memory is not possible[1]. Secondly, CC-NUMA is vulnerable to cache conflicts. Conflicts resulting in the replacement of remote blocks will cause faults on a subsequent access. Had inclusion been maintained, the local memory could satisfy the request with no software overhead. Finally, the total amount of remote caching in the node is limited to the size of the secondary cache.

4.3.1.3 Comparative Hardware Complexity

Of all four solutions, S-COMA requires the simplest hardware additions. It is limited to the state table and some control in an ACD. CC-NUMA comes second in terms of hardware complexity. It requires a state table and a slightly more complex control in the ACD, plus some support for remote write backs and cache fills from remote. COMA and RC-NUMA have the most complex hardware. The state table must be extended to contain tags and some matching logic for the tags must be included in the ACD. However, overall,

1. Of course, we could assume prefetching in the caches but this is a topic beyond the scope of this research.

these mechanisms are very simple, when compared with a full-fledged hardware coherence controller. Last but not least, S-COMA, COMA, and RC-NUMA consume more memory than the CC-NUMA in order to replicate data across nodes.

4.3.2 Software Support

As we have seen, the implementation of software shared-memory raises the problem of choosing the level where the coherence protocol is integrated. S-COMA's handlers must be implemented at the user-level since they rely on the virtual memory system. By contrast, in CC-NUMA and COMA, which rely on physical addresses only, we can implement the software handlers at the kernel level. A kernel-level implementation of a coherence protocol allows the operating system to directly deal with the event without involving the user application layer.

We use the same techniques to implement the coherence handlers of CC-NUMA, RC-NUMA, and S-COMA as we have described for SC-COMA. While waiting for a reply after a miss, the processors simply spin (S-COMA and CC-NUMA) or perform allocation in the attraction memory (COMA) or the remote data cache (RC-NUMA), possibly sending out replacement or write-back requests. The need to perform allocation in a cache, for RC-NUMA and COMA, complicates the MEXC handler and increases its size. The handling of replacements complicates COMA's CINT handler, also increasing its size. The total sizes of the handlers for the four architectures are shown in Table 4.6. Clearly these handlers are still very small and could easily hold in the first level cache of any modern processor.

Architecture	Access Fault Handlers (MEXC)	Network Interrupt Handlers (CINT)
CC-NUMA	304	3,040
S-COMA	348	2,964
RC-NUMA	612	3,132
COMA	1,536	3,600

Table 4.6. Total size of protocol handlers (bytes)

4.3.2.1 Coherence Protocols

We have described the protocol used for COMA previously. The coherence protocol for CC-NUMA is a classic write-invalidate protocol [73]. Invalidations are issued by and acknowledged to the home node. Transactions involve two or four hops. S-COMA and RC-NUMA use the same protocol with only minor modifications. A replacement policy must be implemented in RC-NUMA for the management of the RDC. When a new frame must be allocated in a RDC, an invalid frame is searched first; if the set is full, a random line is picked for replacement.

4.3.2.2 Page Cache Management in S-COMA

In S-COMA, every processor has a finite pool of page frames dedicated to storing shared data. Some of these page frames permanently host data that is placed locally, while others replicate pages placed in remote nodes. The management of this page cache is done by a custom device driver, invoked on every page fault. The page fault handler must obtain a free page frame, initialize the status of the lines in that page to Invalid and update the MMU hardware and system tables with the new mapping. Whenever an (old) page is deallocated to make room for a new one, all blocks from the old page must be flushed in the cache and dirty lines from the old page in memory must be written back to their home. Our evaluations have revealed that an LRU page replacement policy is better than FIFO.

When an access faults at the ACD, only the physical address can be recorded and, since this address is of local significance only, a reverse translation to the global (virtual) address must be performed by the software handlers using a table with an entry for each page frame. This table is filled in by the page fault handler and contains other information, such as the home node for a page.

For S-COMA, we do not perform the detailed simulation of the operating system activity during page faults. When page faults are detected, the actions of the page fault handler are performed by the simulator and we suspend the faulted processor for the estimated duration of this activity. The time penalty has a constant component of 1,000 cycles, accounting for start-up and recovery time, management of the page cache and update of system resources and tables, and a variable component for flushing a deallocated page out of the cache and main memory. On the average, the noticed cost of a page fault is 1100-1500 cycles.

4.3.3 Factors Affecting the Performance

Local transactions in the four architectures are performed at hardware speed. CC-NUMA, S-COMA, and RC-NUMA (for local references) can do the access checking in parallel with the data access. However, COMA and RC-NUMA (for remote references) incur the overhead of fetching and comparing tags. These operations must precede the actual data access because the precise position of the line in a set is unknown. Even when the tag table is stored in a separate memory, the actual line transfer cannot proceed in parallel, unless main memory is wide enough to provide the whole set in one access. This is clearly an option when the AM/RDC are direct-mapped.

Overall, the performance of the four DSMs is largely dominated by the stalls due to remote transactions, which require software intervention. We now discuss several aspects of these software transactions.

Message Latency. Message latency is higher for handlers executed at the user level than for handlers executed at the kernel level. This is especially true when the application processor is multiplexed, because passing an access fault to user-level requires a costly traversal of the operating system. However, for a uniform comparison, we use the same light-weight interruption approach in S-COMA and assume that access faults and external requests from the network interface are always handled in the application address space. Thus, access faults handlers execute almost as if a dedicated protocol processor is present and the application processor is idling. As for interrupts due to external requests, we will see that their overhead is not a critical factor.

Memory Pressure. A marked advantage of CC-NUMA is that shared memory is only needed to store the shared data set. In RC-NUMA and the COMAs, memory must also be provided for replication. When memory pressure increases, replication is more constrained, which causes refetches or ping-ponging of memory lines between nodes. Memory pressure increases the node miss rate and the rate of remote node write-backs. In S-COMA the replication is less efficient than in COMA because of memory fragmentation: some of the lines reserved when a page is replicated are never accessed or are accessed very little but the entire page frame is nonetheless committed. This memory fragmentation introduces *memory pressure inflation* and increases the page fault rate.

Node Hit Rate. In hybrid DSM, remote transactions are very costly, thus the overall node hit rate (fraction of references which complete locally) is critical. In CC-NUMA, there is no caching in the memory. Thus the node hit rate is highly dependent on the cache hit rate and on the data placement in memory. By contrast, the node hit rate is not affected by the cache hit rate in a COMA (because of inclusion). However, since coherence and cold misses in the node are independent of the amount of caching in the node, only the replacement misses (both capacity and conflict misses) that are remote in CC-NUMA can be cut in a COMA [74]. The node hit rate in COMA is affected by memory pressure.

Remote Data Cache versus Attraction Memory. Both RC-NUMA and COMA maintain fine-grain backups of the processor caches in main memory. RC-NUMA's RDC provides a means for remote data to *replicate*, but not to *migrate*. The AM in COMA provides both and, intuitively, it should further increase the node hit ratio. This is especially true when the remote working set overflows the capacity of the RDC, which can happen when data placement is poor or memory pressure is high. The AM has a higher capacity and captures the remote working set by migration. However, there are cases when the RDC provides a higher node hit ratio. When the remote working set fits in the RDC and the application requires mostly data replication, conflicts in the AM could be higher than in the RDC [85]. Finally, RC-NUMA has the advantages that tag-checking is done only for remote references and the tag storage is smaller.

Node Replacements and Write-backs. In RC-NUMA and the COMAs, cache write-backs are always local and done in hardware, whereas in CC-NUMA they can be remote thus creating interruption overhead at the home. Memory and RDC write-backs are executed in software in both COMAs and RC-NUMA. In COMA and RC-NUMA, a line is written back after the request for the new line has been sent out. Thus, replacements (albeit complex) are dealt with while the processor is idle, waiting for the new line. They are not in the critical path of the processor, and only create overhead in the processors that are interrupted by the write-back. This is not true in S-COMA. Memory write-backs occur whenever a replicated page is deallocated. *Before* a page frame can be deallocated, all its dirty lines must first be written back by the fault handler and write-backs occur in bursts. Deallocation in page chunks also creates the false replacement effect [67], i.e. memory lines are written back while they are still actively accessed.

Data Placement Effects. In all four architectures, every page in the shared address space has an associated home node, for which it is a *local* page. In the absence of replication, data from local pages can be accessed much faster than data on remote pages. Even

when page replication is supported, a careful placement of pages can improve performance in two ways. First, memory consumption by replication is reduced when pages are placed in nodes that are guaranteed to access them often. This leaves more room to replicate other pages. Second, the protocol is more efficient by cutting the number of request forwarding that a home node must do. In a hybrid DSM, this also reduces the protocol processing overhead.

Page Faults. This overhead, which is on the critical path of the access, is only incurred in S-COMA.

Effect of Memory Line Size. The memory line size for RC-NUMA and the COMAs may be selected independently from the cache line size. A larger memory line size may have positive or negative effects. In some cases, the prefetching effect in memory cuts down the number of node misses. In others, the cache miss rate may increase because of the back-invalidations required to maintain inclusion.

In order to quantify and compare these effects, we have run execution-driven simulations of six SPLASH-2 benchmarks on the four architectures. In this evaluation, we employ different data placement strategies.

4.3.4 Evaluation Results

We now compare the relative performance of the four hybrid architectures. Figure 4.11 shows the execution times and their components. For S-COMA, RC-NUMA, and COMA, we include measurements taken at several memory overheads: 100%, 50%, and 33%, corresponding to memory pressures of 50%, 66%, and 75%. S-COMA includes an extreme case assuming infinite memory. (At this point, page replacements disappear, but the memory consumption is prohibitive, as indicated in Table 4.11.) On the left side of each plot, results correspond to the round-robin placement; on the right, to the best static placement.

The results are normalized with respect to COMA 50% pressure and round-robin data placement.

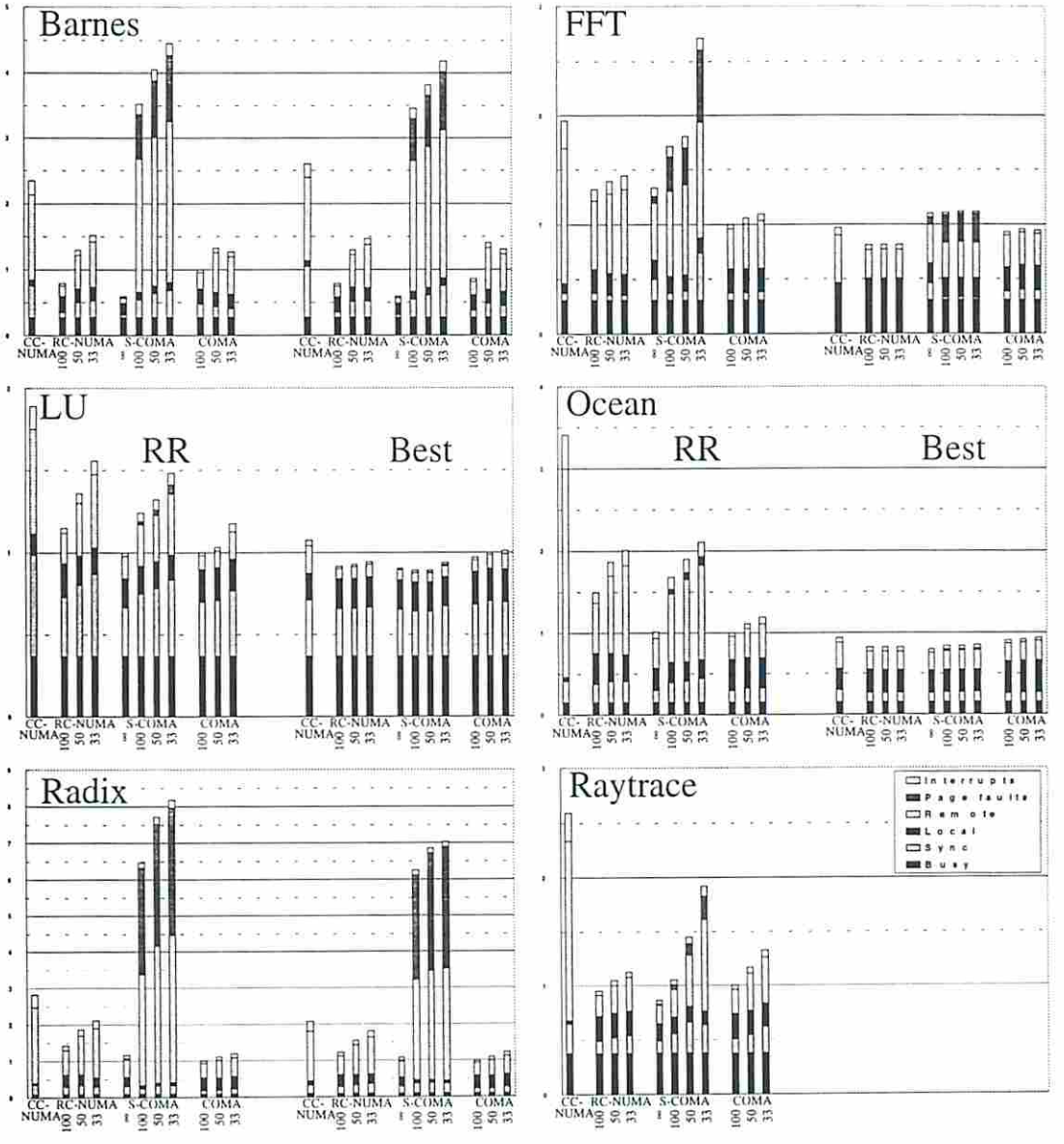


Figure 4.11. Execution times for the hybrid architectures

The execution times are broken down into several components. The *busy* time corresponds to the effective instruction processing time of the processor. The processor is

stalled during *synchronization* events and whenever it needs to access the external cache or main memory. When the access can be completed without requiring the cooperation of other nodes, the delay is counted as *local stall*. In a hybrid system, this corresponds to accesses performed without software intervention. The other accesses contribute to the *remote stall*. In the absence of a dedicated protocol processor, an additional component of the execution time is due to the processing of external requests which *interrupt* the application thread. We do not include in this category the overhead of requests occurring while the processor spins at a synchronization or a pending remote access. Finally, S-COMA has a significant *page faulting* activity.

The characteristics of each architecture directly or indirectly affect each of the execution time components, except for the busy time. The direct effects come mostly from the fact that a cache miss either hits in the local memory/AM/RDC, adding a small contribution to the local stall, or misses, adding a large contribution to the remote stall. Additionally, an access may incur delays due to a page fault in S-COMA. Essentially, the direct effects are a matter of event counts and associated time penalties. Table 4.8 lists the fraction of shared accesses that generate an access fault assuming a best placement. This measure is indicative of the direct contribution of accesses to the local and remote stalls.

The indirect effects are due to contention created by intense protocol overhead and write-back/replacement activity. Contention increases queuing delays of protocol requests, thereby the remote access latency. It affects the load balance as well, hence the synchronization delays. The write-back/replacement activity creates an overhead at the reception and, in S-COMA, at the sender as well. Table 4.9 gives the counts for these events. Finally, when the write-back/replacement unit is larger than a cache block, the cache hit ratio can be affected negatively (because of inclusion).

	Cold+ Coherence	Capacity local (RR)	Capacity remote (RR)	Capacity local (Best)	Capacity remote (Best)
Barnes	0.07	0.08	3.25	0.18	3.15
FFT	0.68	0.06	1.71	1.53	0.24
LU	0.12	0.02	0.49	0.34	0.17
Ocean	0.39	0.17	6.51	6.27	0.41
Radix	1.50	0.31	8.96	3.27	6.00
Raytrace	0.66	0.85	5.50	-	-

Table 4.7. Cache miss ratios (%) for a 16KB 4-way set-associative SLC

Benchmark	CC-NUMA	RC-NUMA			S-COMA			COMA		
		50%	66%	75%	50%	66%	75%	50%	66%	75%
Barnes	3.22	0.21	0.73	0.98	2.79	2.97	3.10	0.29	0.87	0.78
FFT	0.92	0.35	0.35	0.35	0.38	0.39	0.39	0.35	0.36	0.36
LU	0.29	0.06	0.07	0.09	0.06	0.06	0.07	0.07	0.08	0.10
Ocean	0.80	0.35	0.35	0.35	0.36	0.36	0.37	0.32	0.33	0.35
Radix	7.50	1.63	2.54	3.22	9.05	10.0	10.3	1.44	1.61	1.74
Raytrace (RR)	6.16	0.47	0.60	0.73	0.61	1.10	1.88	0.51	0.78	0.95

Table 4.8. Node miss ratio (%) for shared data (Best placement)

Benchmark	CC-NUMA	RC-NUMA			S-COMA			COMA		
		50%	66%	75%	50%	66%	75%	50 %	66 %	75 %
Barnes	661	169	318	373	533	573	617	32	96	139
FFT	0	28	35	36	145	145	145	41	132	116
LU	0	0	0	0	0	0	0	0	1	11
Ocean	1051	0	1	3	54	66	69	0	10	58
Radix	72227	4484	15726	23910	88779	99033	102186	2041	5705	9039
Raytrace (RR)	627	49	45	45	57	48	80	88	230	457

Table 4.9. Write-back or replacement ratios (per 10^6 references) (Best placement)

Benchmark	50%	66%	75%
Barnes	3.70	3.25	3.00
FFT	1.08	1.08	1.08
LU	18.06	15.99	10.37
Ocean	11.77	10.77	10.13
Radix	1.02	1.01	1.01
Raytrace (RR)	7.63	6.18	5.39

Table 4.10. S-COMA average number of block faults between page faults
(Best placement. Block faults include all RD/WR misses.)

Barnes	FFT	LU	Ocean	Radix	Raytrace
22.2	19.5	8.8	3.9	18.8	6.8

Table 4.11. Page replication factor in S-COMA
(ratio between the number of allocated page frames in S-COMA with infinite memory and CC-NUMA)

We now compare the results for each benchmark, paying particular attention to the following factors: fraction of replacement misses in the application, existence of a good data placement for the application, memory hit latency, and prefetch/inclusion effects.

In **Barnes**, we see little effect of the data placement strategy because of the dynamic work scheduling among the processors. According to the SPLASH-2 report [82], the majority of cache misses are capacity related and CC-NUMA is not expected to perform well. The fine-grained AM in COMA can satisfy many capacity cache misses, with limited replacements. RC-NUMA's RDC does it just as well at 50% pressure, because it is large enough to accommodate an important working set WS1. As the RDC becomes smaller, WS1 no longer fits and the node miss ratio picks up. Overall, COMA performs the best, especially when memory pressure is high. This shows the effectiveness of the AM even after tag-checking is accounted for.

Barnes' main data, bodies and cells, are fine-grained and spatial locality is low. As the simulated galaxy evolves from one time step to another, the assignment of bodies to processors and the configurations of the cells change, leading to irregular access patterns.

Consequently, S-COMA performs poorly even at a memory pressure of 50%. The main culprit is page cache thrashing, which creates a large page faulting overhead and deteriorates the node hit rate. Tables 4.8 and 4.9 indicate clearly that replication is so inefficient that, at 75% pressure, S-COMA's node miss and write-back ratios are almost identical to CC-NUMA. The explanation is simple: Table 4.10 shows only three line misses per page fault, indicating that the memory is grossly underutilized (there are 32 lines on a page). Additionally, there is a negative effect on the synchronization stall, because the protocol's high consumption of processing bandwidth induces load imbalances.

FFT is ideally suited for a CC-NUMA architecture. Its cache misses are almost exclusively cold and coherence related. Scheduling of the work is static and, under the best placement, there are no capacity misses to remote data, thus replication is useless. In fact, CC-NUMA would be a winner in this case, if it weren't for the memory prefetching benefits of the 128-byte block in the other systems.

The spatial locality properties of FFT are such that there are absolutely no gains in prefetching blocks larger than 128 bytes, which hurts S-COMA. During communication phases, each processor marches through a large number of remote pages fetching just one 128-byte block from each, thus thrashing S-COMA's page cache. As shown in Table 4.8, the node hit rates are almost equal for S-COMA, RC-NUMA, and COMA; however, the average miss in S-COMA is much more costly because most of them involve a page fault (see Table 4.10).

There are two reasons why COMA, under best placement, performs worse than RC-NUMA. First, processor 0 initializes all the data during the sequential phase, which forces some replacements for local data. When the parallel section begins, processor 0 recovers its local data from processor 1. This creates hotspotting on the two processors and increased synchronization penalty for the first lockstep. Secondly, there is the tag-check-

ing overhead for memory hits. Still, COMA outperforms RC-NUMA under the round-robin placement because the AM satisfies more capacity misses.

LU has a small cache miss ratio, but a good fraction of it is due to capacity. Like in FFT, under best placement, most of these capacity misses can be satisfied by the local memory. Again, CC-NUMA misses out on the prefetching effects of a larger coherence unit and suffers from a higher node miss ratio and increased synchronization delays due to more protocol processing overhead.

The node miss and write-back rates are so tiny that effects of the memory in the other three systems are marginal. The remote working set is small enough to fit in all the main memory caches and spatial locality is sufficiently high not to cause significant page faulting activity in S-COMA. Thus, only coherence misses are dealt with in software, putting COMA at a slight disadvantage as the tag-checking overhead increases the local stall. COMA, however, is a clear winner under round-robin placement, as neither the page cache, nor the RDC, are large enough to store the entire remote working set. In particular, the 33% RDC has a size almost equal to the SLC and conflicts in the RDC start to interfere with the SLC due to inclusion (relaxing inclusion for clean blocks is an efficient way to alleviate this problem).

It is well known that **Ocean** does very well on a CC-NUMA under best placement. The scheduling is static and the spatial locality is large. Ocean has a large number of capacity cache misses but the misses are mostly local under the best page placement, thus incurring no software overhead in CC-NUMA. The performance of COMA suffers slightly from the tag-checking overhead on memory hits.

S-COMA is able to satisfy the small fraction of capacity misses to remote data with little page faulting overhead. This is due to the coarse grain of the data structures, exceeding page size. Memory utilization is excellent. COMA brings a small improvement in the node hit ratio, but pays a higher price for the bulk of local memory hits. Overall, a

fine-grain cache in main memory is not needed in this application. However, note that COMA is again the architecture least sensitive to data placement.

Radix, in addition to a high cold and coherence miss ratio, has significant capacity misses. Because the memory access pattern of each process is data-dependent, the best placement cannot improve the execution time to a point that CC-NUMA is competitive. The poor spatial locality also creates a large number of remote cache write-backs, increasing the fraction of time spent in interruptions and synchronizations.

Radix is a disaster for S-COMA. The reason is the permutation phase where, even with the best placement, an overwhelming fraction of writes are to non-local data. Because accesses are scattered in this phase and the data structures to be updated are fine-grained, a large number of pages have to be mapped in the page cache and subsequently deallocated to make room for others. In effect, a single line is used throughout the lifetime of a such page (see Table 4.10). The performance of S-COMA can only be improved at the cost of much higher memory consumption: for the 0% pressure point, almost 19 times more memory is necessary than in CC-NUMA.

Radix is one of the best arguments for COMA against RC-NUMA because Radix requires support for data migration, and not just replication. Thus, the larger AM satisfies more capacity misses than RC-NUMA's RDC.

Raytrace benefits mostly from the replication of the read-only scene, the main data structure. Work scheduling is dynamic and accesses through the scene data are input-dependent. Thus no best placement was recommended by the programmer. There are significant capacity misses and, again, the 128-bytes coherence unit adds an additional prefetching benefit. Both reasons disadvantage CC-NUMA.

Fine grain objects and little spatial locality make replication less effective in S-COMA, especially at high memory pressure. Because Raytrace only needs data replication, RC-NUMA has an advantage over COMA, as the RDC can achieve replication with

fewer conflicts than the AM [85] (again, relaxing inclusion helps COMA). The node miss ratios (Table 4.8) indicate this clearly. This observation is corroborated by the large number of replacements in COMA (Table 4.9), as data gets shuffled around when the attraction memories try to accommodate replicated lines.

4.3.5 Discussion

CC-NUMA's advantages are relative hardware simplicity, parsimonious utilization of memory, and fast access for local memory hits. Whether CC-NUMA is competitive depends mostly on the number of remote capacity misses in the cache and the optimal size for the coherence unit. CC-NUMA's performance is very sensitive to the data placement. The placement cannot be improved significantly when the scheduling is dynamic or when the data access pattern has very little spatial locality. Overall, CC-NUMA cannot best COMA or RC-NUMA even at 75% memory pressure, especially if programmers are oblivious of the idiosyncrasies of the machine.

The hardware of S-COMA is even simpler than CC-NUMA and the local memory hit access is just as fast. However S-COMA's consumption of memory is prohibitive for applications with fine-grain sharing because of fragmentation caused by the coarse grain of allocation in the page cache. As a result, under finite memory constraints, the node miss rate and page faulting overhead soar. Table 4.10 shows the average number of memory misses between two page faults. This number is indicative of fragmentation, being an upper bound on the number of blocks touched during the lifetime of a page frame. Except for Ocean and LU, there is obvious fragmentation. By reducing the usable capacity of the main memory cache, fragmentation directly increases the node miss ratio. The page faulting overhead is very significant in some cases. (Two previous studies either assumed a very low penalty for a page fault [67] or very low memory pressure [63].) There is no real indication from our results that S-COMA is effective at dealing with fine-grain sharing in

the general case, unless large amounts of memory can be used without concerns for efficiency (Ocean and LU have coarse grain sharing and would probably perform well on a pure software DSM). In fact, even with twice as much memory, S-COMA does not seem to be a clear winner against CC-NUMA with good data placement.

Due to the reduction of remote capacity misses, RC-NUMA is always an improvement over CC-NUMA. Additionally, it solves some of its technical problems. The RDC captures remote cache write-backs, enables prefetching by using larger inter-node coherence units, and provides an immediate storage for incoming blocks. The cost is the complexity of the RDC controller (similar to COMA's AM controller) and its overhead on remote misses. Compared to COMA, RC-NUMA requires less tags because the RDC is smaller than the AM. However, the tags are longer and could make the packing of the Tag and State table less efficient. The performance of RC-NUMA is still inferior to COMA in situations when there are benefits from fitting a larger working set in the AM. This is especially true when data migration is required either by the access patterns of the application or to correct the effects of bad data placement.

Overall, COMA shows consistently good performance across benchmarks and data placement strategies and is only narrowly beaten by other machines when applications exhibit coarse-grain sharing and high spatial locality, mostly because of the slower memory access time on a hit. The hardware cost is somewhat higher because of the AM controller and the tag storage. We note that COMA is much more stingy in memory consumption and less sensitive to memory pressure than S-COMA. Even at 75% pressure, COMA does very well. We should also point out that inclusion relaxation further improves the performance of COMA in a significant way. This is true for RC-NUMA, as well, but to a lesser degree, while S-COMA can not take advantage of this technique.

4.3.6 Conclusions

In hybrid DSM systems coherence among nodes is maintained in software but hardware support (the Access Checking Device or ACD), is added to reduce the size of the coherence unit to a small block to avoid false sharing, thus supporting fine-grain sharing. This study makes the first comparative performance evaluation of four hybrid architectures: CC-NUMA, RC-NUMA, S-COMA, and COMA, by using execution-driven simulation of detailed implementations on a common platform. Basically, we show that COMA achieves the best (or close to the best) performance level across various applications with widely different behaviors. In particular, the performance level of COMA is independent of the data placement and is very good across various memory pressures. This is in contrast to CC-NUMA and S-COMA where performance exhibits large variations across applications and, thus, the programmer must be aware of the memory organization in order to attain acceptable performance levels. RC-NUMA sometimes performs marginally better than COMA under the best data placement, but is more sensitive to data placement and memory pressure. Overall, COMA realizes the goals of ease of programming, high performance, and consistent behavior across applications and data placements.

As shown by our study, coarse allocation in main memory caches may defeat the purpose of supporting fine-grain sharing because of overheads induced by fragmentation. Thus, we advocate to either avoid replication by implementing a hybrid CC-NUMA, or better yet, to replicate memory in units of coherence blocks -- not pages-- and manage (part of) the memory as a set-associative cache by implementing a hybrid RC-NUMA or COMA. COMA and RC-NUMA require identical hardware resources to support the access and management of the set-associative RDC/AM. The difference is made by the format of tables and the address ranges. It is easy to incorporate programmability in the ADC in order to configure it for one architecture or the other. Even better, the two memory

organizations could coexist and adaptive algorithms could be used to migrate data between local memory and attraction memory to better suit all possible data sharing patterns. R-NUMA [27] is a similar proposal of integrating RC-NUMA and S-COMA and it, too, could be implemented as a hybrid system.

Compared to hardware DSM systems, which are trying to balance access latency, miss rates, and implementation complexity aiming at a high-performance, yet cost-effective and timely design, hybrid systems have a single measure of value: the overhead caused by handling node misses in software. This shifts the preference toward the more complex memory organizations and coherence protocols, even at a small penalty for the access latency. As hybrid DSMs are slower than hardware DSMs, the hybrid approach must offer other advantages such as reduced costs, ease of development, and flexibility. The ACD mechanism in the memory controller is a simple design, which can be standardized across generations of machines. In this context, a small performance slowdown can be acceptable if it allows constructors to design, debug, build, test, and manufacture large multiprocessors using the latest processors with a shorter time to market. The ACD mechanism can also be attached to the bus of a commodity workstation and used as a snooping device [63]. In this study, we have used a write-invalidate protocol, and thus we have not explored the additional performance improvements possible with customized or adaptive protocols.

In the current implementation, we have assumed that the processor can easily restart an access faulted at the local bus level. This may be difficult with high-performance processors, especially in the presence of non-blocking loads. Non-blocking stores are easy to handle, provided access to the store buffer, so that a faulted write can be recovered. Since store buffers are flushed in the events of a trap or interrupt, it is guaranteed that write faults cannot occur in unexpected situations. We will later investigate the performance impacts of using a release consistent hardware.

The use of a dedicated protocol processor solves the access restartability problems. Apart from implementation advantages, the performance trends and effects observed in this study would hold in a machine with protocol processors as well. The node miss rates are largely unaffected, leading to the same number of accesses that need software intervention. The latency of these accesses would improve slightly, by eliminating the costs of context switching. The overhead of interrupts would be eliminated as well, but we have seen that this is not significant. Furthermore, the issue of how to replicate data at the memory level is independent of the presence of a dedicated protocol processor. S-COMA would still have a lower node hit ratio and the overhead of page faults. We, therefore believe that a set-associative memory cache brings the same performance advantages, even in the presence of a protocol processor.

Chapter 5

OPTIMIZATIONS FOR SC-COMA

5.1 Relaxed Inclusion

Data replication in the attraction memories is less efficient than replication in the processor caches. The reason is that the attraction memory sets must accommodate other data, which is not part of the working set, thus having less room for replication. By contrast, the processor caches can discard (or send to the lower memory level) blocks that are not part of the working set, thus being able to utilize all the frames for needed data. True, data can be evicted from an attraction memory through replacements. However, for wide sharing situations, it is sometimes impossible to find enough frames in a set to satisfy the replication needs for particular blocks. Even for more common sharing patterns, at high memory pressure, the expansion in the sharing degree of a particular block may force, at the attraction memory level, a decrease in the degree of another which is still part of the working set and, possibly, cached. Conflicts in the attraction memory do not necessarily imply conflicts in the cache, as the two caches might be indexed differently (e.g. physically in the attraction memory and virtually in the processor cache).

An attractive solution to this problem is to relax the requirement that a block present in the cache must have a frame allocated in the attraction memory (the inclusion property) [40][41]. Keeping inclusion is still desirable for blocks dirty in the cache as they will generate cache write-backs when victimized. If no frame is allocated in the attraction

memory during a write-back, SC-COMA's level of hardware support is unable to handle the situation autonomously (similar to problems we have seen for the hybrid CC-NUMA). However, inclusion can be relaxed for clean blocks, which constitute the majority anyway.

Inclusion is relaxed in SC-COMA in the following two situations when lines in state Shared are victimized in the attraction memory:

- During allocation in the attraction memory for a requested line, a Shared block is victimized.
- During the processing of a replacement request the line is accepted onto the frame occupied by a Shared line.

Overall, inclusion relaxation is very effective in reducing the node miss ratio and the rate of replacements, especially when the memory pressure is high and/or the applications exhibit wide sharing patterns. In the context of software-implemented coherence, inclusion relaxation also comes as a simplification and reduces the size of the protocol handlers and the software-overhead. Indirectly, this reduces slightly the average miss latency (not directly because the code for maintaining inclusion is not on the access critical path).

Figure 5.1 illustrates the benefits of relaxing inclusion by showing the "before and after" execution times for the SPLASH-2 benchmarks and a 75% memory pressure. In fairness to the hardware implementation, we have relaxed inclusion in HW-COMA as well. Each of the two execution times for SC-COMA are normalized with respect to their equivalent HW-COMA. In absolute terms, the execution time of SC-COMA improves by an average of 16.8% after relaxation. Barnes has the highest improvement (48.2%), due to significant conflicts in the attraction memory. LU follows with 18.1%, while Raytrace and Radix also see two-digit improvements. Relaxation achieves more modest benefits for Ocean (6%) and FFT (4.2%).

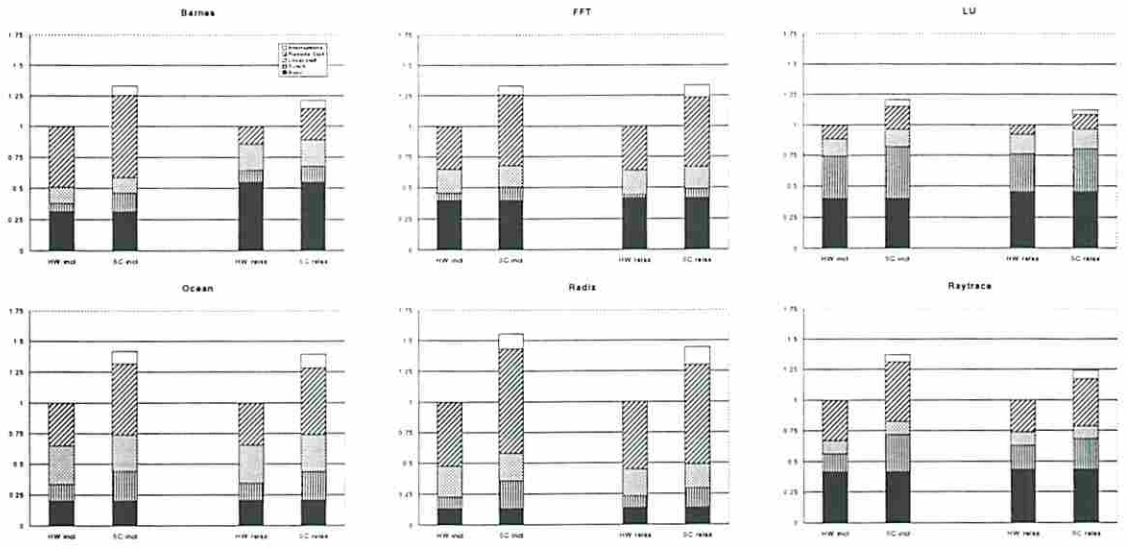


Figure 5.1. Execution times before and after relaxing inclusion

Execution times correspond to 75% memory pressure. The leftmost two bars are for HW-COMA and SC-COMA before relaxing inclusion. The rightmost two bars present the same information after relaxing inclusion. Each SC-COMA time is normalized with respect to its HW-COMA counterpart.

Benchmark	Node miss ratio (%)	
	Before relaxation	After relaxation
Barnes	0.761	0.169
FFT	0.512	0.504
LU	0.150	0.082
Ocean	0.559	0.540
Radix	1.807	1.824
Raytrace	0.972	0.774

Table 5.1. SC-COMA node miss ratio before and after relaxing inclusion

To better understand the effects of relaxing inclusion, Table 5.1 gives the node miss ratios before and after relaxation. As can be seen, Barnes and LU both show significantly smaller miss ratios after relaxation, explaining the much reduced remote stalls from Figure 5.1. These two applications have data structures that are widely shared. Due to dynamic work assignments, Barnes also has varying degrees of sharing to much of the entire data set.

Raytrace has significant amounts of read-only data that starts to replicate selectively and in moderate amounts throughout the attraction memories. Thus, relaxation is beneficial at a lesser degree and mostly because the processor cache acts as an extra capacity to the node caching space.

In Ocean only the nearest neighbors share data and very little is widely shared. Most of the working sets quickly migrate to their respective user nodes where they get pinned down in the attraction memories through mastership, thus avoiding victimization during allocation after a miss. At the same time, the remaining remote working set exhibits limited capacity misses, thus it makes no impact whether victimization in the attraction memory is forcing eviction from the cache or not. Similarly, in FFT, there is little need for replication, thus very little can be improved in the miss ratio.

Radix has practically no wide-sharing and Table 5.1 confirms by showing practically unchanged node miss ratios. However, Radix does show a smaller execution time after relaxation. A closer investigation revealed that the improvement is actually due to smaller average latencies (by almost 10%). Most of the victimizations in the attraction memories and the replacements happen during a bursty all-to-all communication phase when the protocol engine is very solicited and queueing delays for requests are high. Because the handlers' execution time is shorter after relaxing inclusion (although just about 5%), the overall execution time is less due to three effects. First, the direct overhead during the processing of replacements is smaller. Secondly, queuing delays are smaller due to reduced occupancy for the protocol engine. This is the most important factor for the reduction in the average latency. Thirdly, lower average latencies have a positive impact on the synchronization delays as load imbalances due to protocol activity have less weight.

Benchmark	Replacement rate (per 10^6 references)	
	Before relaxation	After relaxation
Barnes	138.5	114.4
FFT	747.3	679.2
LU	419.4	242.7
Ocean	2323.6	2215.3
Radix	7402.8	7786.6
Raytrace	429.2	422.4

Table 5.2. SC-COMA replacement rate before and after relaxing inclusion

Table 5.2 is listing the replacement rates before and after relaxation. As can be seen, there is a smaller impact than the one we have seen for the node miss ratio. Replacements occur during allocations in attraction memory sets where all the lines are in state Master-Shared or Exclusive. Relaxation only allows caches to keep blocks from a line evicted in state Shared and does not contribute to reducing the "mastership pressure" on attraction memory sets. However, by reducing the node miss ratio (especially read capacity misses that would fetch Shared lines), relaxation does allow the attraction memory sets with high mastership pressure to avoid frequent allocations, thus replacements.

It would be interesting to look at the effects of relaxation in the context of a protocol where mastership for clean line is passed on the last reader [85]. Intuitively, this creates more opportunity for "harmless" victimization of clean lines in the attraction memories. However, our general experience with this kind of protocol was that, although the node miss ratio can be smaller, there are more replacements, especially for read-mostly data (as would be the case in Raytrace).

5.2 Optimizing the Replacement Algorithm

The quality of the replacement algorithm in COMA is reflected by two measures: the node miss ratio and the replacement rate. The node miss ratio directly translates into remote stalls, while the rate of replacements contributes to protocol engine occupancy and net-

work traffic. A hardware protocol implementation (with lower protocol engine occupancy) would probably emphasize more a low node miss ratio, but in a software implementation, the two measures are equally important.

The replacement algorithm can be evaluated individually at the level of each *global set* in the attraction memories. The global set is the collection of all the attraction memory sets with the same number, as shown in Figure 5.2. Assuming the set-hashing function is node-independent, the global set is an invariant. Thus, a line is confined to a global set even after replacements. A global set contains a constant number of lines in master states, as set by the data allocation requirements, along with a variable number of lines in non-master states: Shared and Invalid. Unfortunately, the design of an optimal replacement algorithm at the global set level is a difficult task, which is partly why this issue was not investigated so far. Another reason has to do with the fact that information about the global set is not available in a single node for the common attraction memory organizations (also see Section 3.5.1).

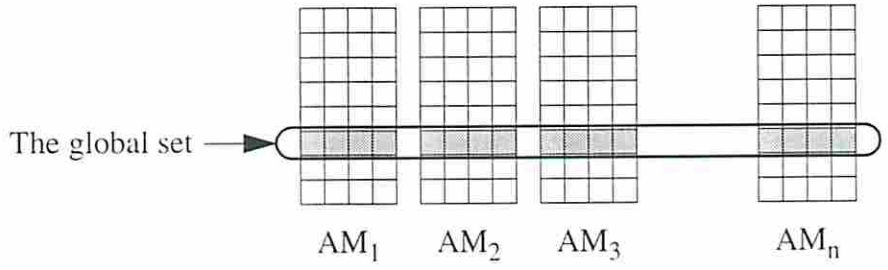


Figure 5.2. The global set in the attraction memories

There are two policies in the replacement algorithm. The *victimization policy* chooses a candidate for eviction from an attraction memory during either allocation after a miss or the processing of a replacement request. When the node has mastership of the victim, the *injection policy* decides the appropriate sequence of nodes which will be

requested to accept mastership of the line. The optimizations described next focus on the injection policy. We assume throughout that the victimization policy is the one described in Section 3.1.5.

A good injection policy is one that has high chances of acceptance at the first node where a replacement request is sent. Recall that replacement requests can be rejected when the set is filled with lines in master states. After a rejection, the replacement is retried. The goal of the injection policy is to keep the number of rejections/retries as low as possible. Note that this goal alone does not guarantee an optimal replacement algorithm. In general, there might be multiple candidates for a "good" first-time replacement, but the overall node miss ratio might be minimized by a single choice.

Before presenting the optimized injection policy, let's explain why the original injection policy was flawed. Recall from Section 3.1.5 that, after a rejection or when the replacement originates at the home node, a request was sent to the immediately adjacent node ($n+1$). This design was based on the assumption that replacements are uniformly distributed over all the nodes. In reality, during initialization performed by the master node (0) in the sequential portion of a parallel program, the master node's attraction memory accumulates a lot of unnecessary data. At the same time, data allocated on the master node is displaced through replacements to nodes 1, 2, and so on, in a decaying proportion. When the parallel section begins, the attraction memories are polluted in an uneven way, which creates protocol activity and synchronization imbalances. Of course, if the parallel section is much longer compared to the time it takes to "stabilize" the attraction memories, this effect would not be so important. However, for the benchmarks we have evaluated, this is often not the case. To eliminate this effect, whenever forced to make an arbitrary choice, the protocol uses a pseudo-random destination node for the replacement.

The injection policy must make its decisions based solely on information available at the local node. Had information about the global set been available, an Invalid or Shared

line could easily be singled out as a sure candidate to accept the replacement. Consequently, our optimized injection policy uses heuristics in the form of two replacement hints.

For lines in state Master-Shared, the injection policy picks a node that is currently in the copyset of the line. This is made possible by the fact that SC-COMA's coherence protocol keeps the copyset at the current master, and not the home node. Because the protocol is non-notifying, the destination of the replacement might have dropped out of the copyset silently. However, the chances are high that it has not and that mastership will be passed on in an ideal way. For the case when the replacement is rejected, retries are made to other nodes in the copyset.

For lines in state Exclusive, the replacement hint points to the node that provided this line, i.e. the previous master. The hint is stored in the same space as the copyset, which is now superfluous. The rationale for using this hint is that the former master might have an invalid copy of the line after having provided the exclusive copy to the current master.

The coherence protocol must be adapted slightly to the use of replacement hints. Recall from Section 3.1.5 that replacements were always sent to the home node first. This assumption is now broken. For the rare situation of a crossover between write and replacement requests (see Figure 3.4), the original protocol was relying on the correct sequencing at the home node of the replacement request and the subsequent NACK sent by the stale master in order to buffer and forward the replacement. Now that replacements can be sent first to a node other than the home node, the home node must be prepared to handle a NACK previous to the reception of a replacement acknowledgment from another node. Thus, an early NACK must be buffered. However, this is not a significant complication and the feature might be desirable in case the network does not guarantee in-order delivery

of messages for a point-to-point connection. It is not clear whether an alternative solution, NACKing directly to the requester node, is any simpler or achieves lower performance.

The following results are based on a trace-driven evaluation of an architecture using SC-COMA's coherence protocol. The reason we use trace-driven simulation is to be able to compare the effectiveness of replacement hints against a replacement algorithm based on the global set information. We consider this latter algorithm to be the best (although it is not optimal) as it always injects replacements with 0% chances of rejection. The global replacement algorithm uses an injection policy at the global set level similar to the victimization policy at the set level. Invalid frames are selected preferentially; otherwise, one of the Shared lines is picked as the target for the injection. Of course, if there are Shared copies of the replaced line, mastership is given to one of them. Ties are broken in a random manner. A trace-driven simulation also allows us to quantify the effects of different replacement algorithms free of any interferences and changes of execution path from one run to another.

Figure 5.3 shows replacement ratios for three injection algorithms under both strict and relaxed inclusion at a 75% memory pressure. The first group of three bars corresponds to strict inclusion; the remaining three bars are for relaxed inclusion. Within each group, the first bar is for the base injection algorithm using randomization, as per our previous observation. The second bar corresponds to an injection algorithm using replacement hints. The third bar illustrates results for the global replacement algorithm. The replacement rates are normalized with respect to the global replacement algorithm with relaxed inclusion (sixth bar). Each replacement rate is broken down into four categories. Two categories are for replacements of Master-Shared and Exclusive lines which are accepted at their first destination, thus without generating retries. The other two categories correspond to all the retries for the two kinds of lines. The effectiveness of hints can be evaluated by the achieved reduction in the amount of retries.

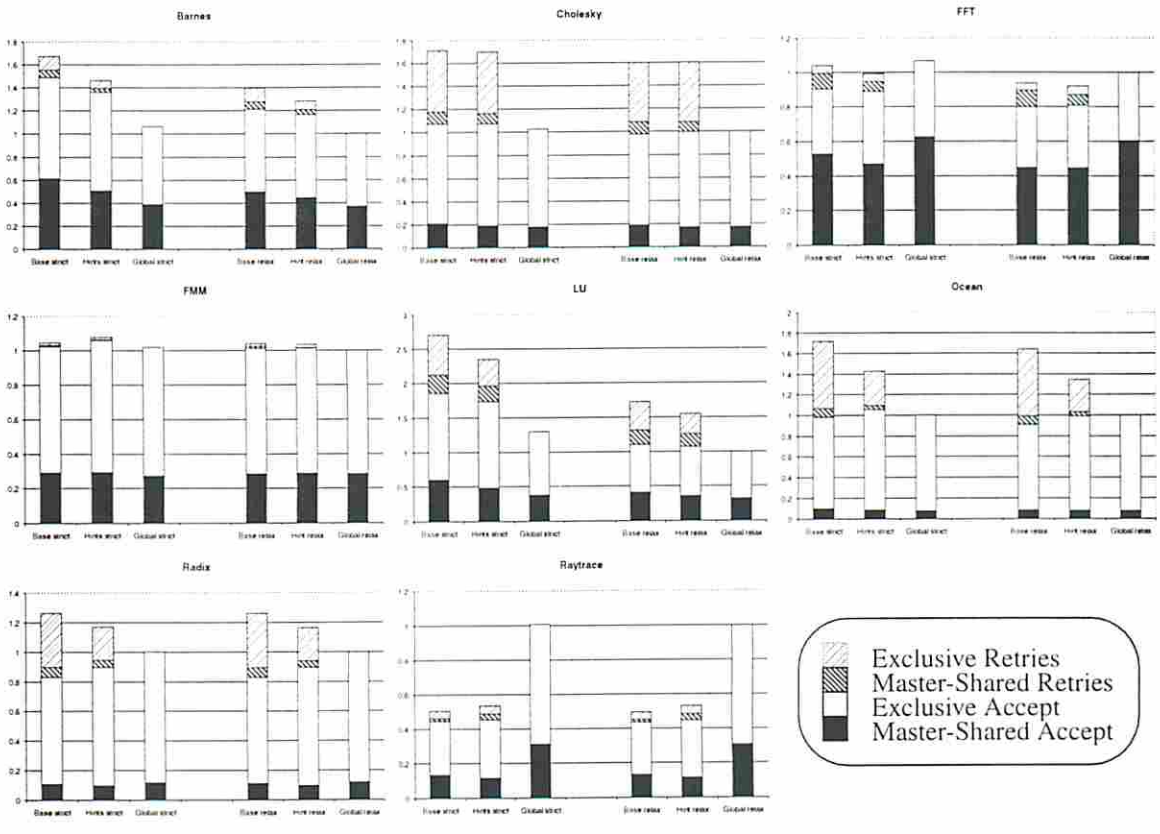


Figure 5.3. Replacement rates for three replacement algorithms.

The first three bars are for strict inclusion; the other three are for relaxed inclusion. Within each group of three, the first bar corresponds to the base random injection algorithm. The second bar reflects the use of replacement hints. The third bar is for the global replacement algorithm which uses global set information. The results are normalized with respect to the sixth bar: global replacement algorithm and relaxed inclusion. Throughout, the memory pressure is 75%.

As indicated by the results, the use of replacement hints almost always generates an improvement over the base injection algorithm. Overall, the effectiveness of hints for Master-Shared lines is less spectacular than expected. A possible reason is that many nodes which receive a replacement request, because the copyset indicates that they have acquired in the past a Shared copy of the line, no longer have the line and cannot accept the replacement. The chances of such an event should be lower for widely shared data. Indeed, Barnes does show significantly fewer retries for Master-Shared lines after using

hints. Furthermore, in Barnes, there is a decrease in the total number of replacements for Master-Shared lines, because mastership is passed to a node that has high chances of keeping the line as part of its working set.

Hints for Exclusive lines always show some improvement over the random injection algorithm. This is fortunate because replacements of Exclusive lines dominate in the large majority of applications. Passing the mastership of Exclusive lines to the former master seems to suit several artifacts. Firstly, in situations of false sharing, mastership can be efficiently assumed by any of the processors involved, as they should have an Invalid frame available (e.g. Radix). Secondly, when the mastership pressure for a set is high (i.e. the set must accommodate more lines in master states than allowed by the associativity), the node will likely go through several rounds of replace-and-fetch for as long as the lines are actively used. In this case, it makes sense to use for each line a single node as an overflow and to consistently send replacements to that node. The chances of acceptance are high and the disturbance created by sending replacements to several nodes is avoided.

The global replacement algorithm shows that there is still room for improvement even after using replacement hints. This is especially true for Cholesky and LU. A look at the results plotted in Figure 5.4 reveals that Cholesky and LU are among the applications with the least amount of frame reuse during replacements (i.e. the injection is done over a frame occupied by a shared copy of the line or has been previously occupied by the line and later invalidated). Replacement hints are targeting exclusively frame reuse opportunities.

The results for Raytrace are quite surprising at first sight. However, one should not equate the replacement rate with overall performance. In an attempt to reduce the node miss ratio, the global replacement scheme protects shared data and eagerly pursues invalid frames to inject replacements (see the high fraction of "Other INV" for Raytrace in Figure 5.4). Indeed, the node miss ratio drops slightly below the ratios achieved by the

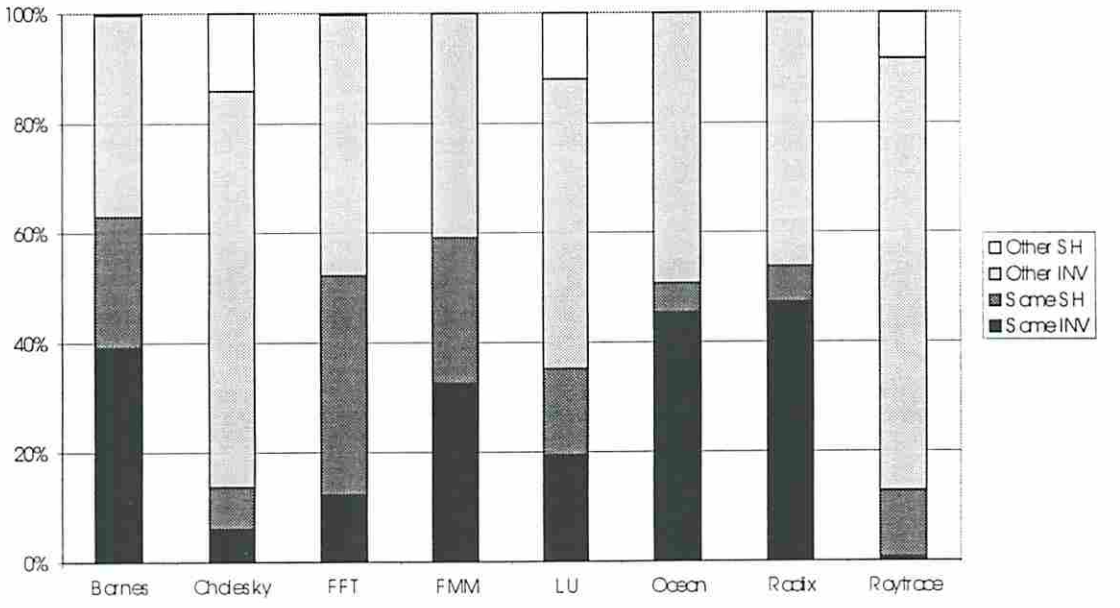


Figure 5.4. Breakdown of injection types for the global replacement

We categorize four types of frames that are selected to host a replacement. The frame could be tagged with the same address as the replaced line, hence the "Same/Other" attribute. The frame could also be in state Invalid or Shared. When looking for a spot in the global set to accept the replacement, preference for these categories decreases in the following order: Same Shared, Same Invalid, Other Invalid, Other Shared. A random frame in the preferred non-empty category is selected as the target of the injection. The memory pressure is 75% and inclusion is relaxed.

other two replacement algorithms (not shown here). However, it appears that this approach injects lines into nodes that begin to suffer from high mastership pressure shortly after the injection. At a high level, such situations could occur during the execution of updates in a critical section. While performing the updates, a node accumulates Exclusive lines, thus increasing its mastership pressure and possibly replacing other lines over the invalid frames of the updated data. When the critical section is entered by the next node, the lines replaced by the first node and possibly accepted by this next node will again be replaced. It is also possible that such a phenomenon is amplified by our use of queue-based locks in the trace collection, which might foster stable patterns of sequencing through the critical section. By contrast, a random injection policy will likely victimize a Shared line in an arbitrary node, thus avoiding mastership hotspots.

5.3 Mastership Hints in SC-COMA

In COMA, the mastership for a line "migrates" to one of the nodes which have the line in their working set. Due to the round-robin data placement, often times the master and the home node for a line are not one and the same. The consequence is that most remote accesses complete in three hops as the home node must be used as an intermediary to forward the request to the current master. This increases the average remote latency and, in SC-COMA, also creates direct overhead at the home node. Had the local node been able to guess the identity of the master, the request could be sent directly, thus bypassing the home node. If the guess is correct, the access completes in two hops; if it is wrong, the protocol must fall back on the original scheme, thus incurring some overhead and further increase in latency.

Mastership hints have been proposed [40][7] to alleviate the three-hop problem. However, evaluation in the context of hardware protocols [7] revealed that the added implementation complexity was not rewarded by performance benefits, at least not in the general case. The situation is likely to change in a software protocol: the implementation costs are null, the potential benefits are higher, due to both improved latency and reduced overhead, and there is the flexibility to enable this feature only for applications where the mastership hints are correct in a high proportion. Two kinds of hints are considered in this study: invalid hints and shared hints [7].

Invalid hints are based on the assumption that the current master of a line is the node that has invalidated the line in the most recent past. Thus, every time a node is processing an invalidation request (or a write request to an exclusive line), the identity of the node requesting the invalidation is recorded as a hint. The hint is stored in the space reserved for the copyset, thus with no extra overhead. Invalid hints work well in producer-consumer communication patterns (e.g. Ocean), where a read miss at the consumer node,

caused by a data update, is always satisfied by the producer node. This hint should also be effective for write misses in false sharing situations (e.g. Radix). SC-COMA uses invalid hints for read misses only. The reason is that we rely on the write request sent to the home node to serialize the writes. A write request sent directly to the current master would complicate the coherence protocol and still require processing at the home node to update the master pointer.

Shared hints are based on the assumption that the current master of a line is the node that has provided the line in the most recent past. This hint works for applications with migratory data, when the pattern of migration is fixed, as well as for cases of producer-consumer communication. In addition to read transactions, this hint to can also be used to optimize ownership transactions, i.e. for writes on clean blocks. In SC-COMA, a shared hint could be stored for Shared blocks in the same storage as the copyset (which is available because only Master-Shared lines have a copyset). The hint is initialized after every read miss. For the same reasons explained for invalid hints, SC-COMA does not use shared hints to optimize ownership transactions.

Mastership hints require some storage. Using the copyset space associated with Invalid and Shared frames is an overhead-free solution. This is also a solution that is very convenient because of the very low execution time overhead to find the hint. The downside is that the storage associated with Invalid frames is volatile and hints are lost after the frames are reused for other lines. Depending on the application, memory pressure, and, possibly, the data allocation, the lifetime of a hint associated with an Invalid frame might be long enough to make the hint available at the time of the next miss to the line. In particular, a high memory pressure reduces the lifetime of such hints. It is uncertain that the use of separate storage for hints in order to extend their lifetime and applicability would pay off for a significant proportion of applications. Additional to the storage overhead, there is

time overhead to search and maintain hashing tables for the hints. The success rate of hints would have to be very large to pay off this overhead.

Tables 5.3-5.5 present results quantifying the effectiveness of both invalid and shared mastership hints. These results are obtained through trace-driven simulations of SC-COMA's protocol at 75% memory pressure, with relaxed inclusion, and using replacement hints. The first two tables contains two kinds of ratios for read and write misses: the ratio of misses that have a hint (hint presence ratio) and the ratio of correct hints when a hint is available (the hint correctness ratio). To see the overall potential of the hints, we indicate ratios assuming an infinite dedicated storage for the hints. For a more realistic situation, we also present results accounting for the volatility of hints associated with Invalid frames when using the copyset storage. The third table indicates, for ownership misses, how often a hint is not necessary (because the miss occurs at the owner) and how often a shared hint (used by a non-master node) is correct.

The results for read misses indicate that there is potential for significant gain from the use of invalid hints for FMM and Ocean. This is likely due to producer-consumer communication patterns. Even when infinite storage for hints is assumed, there is little reason to consider the use of hints in other applications. In some cases, the applicability is low due to cold misses (FFT) or a long sequence of capacity misses for private or read-only data (LU, Cholesky, Raytrace)[1]. In other cases, the success rate of invalid hints is not very high because of migratory data (Barnes).

Shared hints for read misses have a higher range of applicability, assuming infinite storage. Overall, they are correct in a high proportion. In some cases, such as Barnes and Ocean, the hints could practically wipe out three-hop misses. Unfortunately, their applicability is just as low as invalid hints, when the copyset storage is used.

1. Read-only data includes data written once during initialization by the master processor.

Benchmark	Invalid hints				Shared hints			
	Infinite storage		Copyset storage		Infinite storage		Copyset storage	
	Present	Correct	Present	Correct	Present	Correct	Present	Correct
Barnes	85.23	64.34	9.35	93.84	96.40	88.20	9.35	81.46
Cholesky	26.39	70.81	18.98	77.31	32.65	52.96	18.98	45.39
FFT	29.25	99.32	0.36	100.00	40.49	33.97	0.36	5.15
FMM	87.32	91.39	75.67	92.91	86.81	86.27	75.67	86.15
LU	4.36	97.98	0.20	100.00	22.64	78.02	0.20	0.46
Ocean	80.16	97.58	48.72	98.48	96.30	96.37	48.72	98.38
Radix	10.37	90.72	2.58	96.48	39.29	92.44	2.58	80.34
Raytrace	4.26	70.17	2.39	77.45	51.43	93.54	2.39	22.52

Table 5.3. Mastership hints presence and success ratios (%) (read misses)

Benchmark	Invalid hints				Shared hints			
	Infinite storage		Copyset storage		Infinite storage		Copyset storage	
	Present	Correct	Present	Correct	Present	Correct	Present	Correct
Barnes	88.69	78.88	28.93	97.55	95.65	72.64	28.93	82.47
Cholesky	20.31	35.29	10.90	55.93	25.09	31.76	10.90	17.12
FFT	0.00	-	0.00	-	21.62	53.80	0.00	-
FMM	72.96	90.66	62.77	91.25	63.77	76.22	62.77	69.62
LU	0.00	-	0.00	-	96.53	2.16	0.00	-
Ocean	0.54	3.23	0.44	2.39	69.62	85.43	0.44	3.98
Radix	41.87	88.43	29.53	93.55	54.08	83.41	29.53	70.11
Raytrace	28.12	100.00	28.12	100.00	65.62	85.71	28.12	66.67

Table 5.4. Mastership hints presence and success ratios (%) (write misses)

Although SC-COMA's current protocol cannot apply mastership hints for write misses, we are indicating the effects of these hints in Table 5.4 to understand whether effort should be spent in this direction. The raw count of write misses is (much) smaller than read misses in all applications, with the exception of Radix, making hints for writes less important. As can be seen, FMM is the only application with certain gains from the

use of invalid hints. Potentially, Barnes, Raytrace, and especially Radix, could benefit also. Shared hints look promising when the infinite storage is used, but fall short of expectations when using the copyset storage.

Benchmark	At master	At non-master & shared hint is correct	Total ownership requests able to bypass the home node
Barnes	30.10	69.56	99.66
Cholesky	13.15	86.84	99.99
FFT	50.52	48.70	99.32
FMM	44.83	55.15	99.98
LU	5.74	93.87	99.61
Ocean	83.18	16.80	99.98
Radix	60.80	39.20	100.0
Raytrace	1.36	98.44	99.80

Table 5.5. Breakdown of ownership misses (%)

Table 5.5 indicates that ownership misses could almost always be handled without the involvement of the home node. When the miss occurs at the master node, invalidations can be sent right away, because the copyset is available. When the miss happens at a non-master node, the shared hint almost always points to the current owner (not always, because of mastership transfers during replacements). Because of the high success rate and the relatively large proportion of ownership misses at a non-master, it might be useful to modify SC-COMA's protocol for ownership transactions. The highest benefits are expected to be seen in applications with a significant proportion of ownership misses: FFT, FMM, Ocean, and Cholesky, in this order.

Overall, mastership hints have good prospects only for selected applications. Given the flexibility of a software implementation, this is not an impediment for providing this feature in the coherence protocol. Also, mastership hints using the copyset storage

should perform better when the memory pressure is low. Thus, hints could be selectively used based on application characteristics and memory pressure.

Chapter 6

EXTENSIONS OF SC-COMA

This chapter investigates the implementation issues and the performance consequences of building SC-COMA on top of other hardware platforms. First, we will look at clusters of symmetric multiprocessors (SMP) where SC-COMA's software inter-cluster protocol must be combined with a hardware intra-cluster protocol. Then, the assumption that the hardware is enforcing a sequential consistency memory model is removed, as non-blocking processor writes are supported by store buffers.

6.1 SC-COMA on SMP Clusters

Symmetric MultiProcessors (SMP) have emerged in the last few years as one of the most attractive basic blocks for larger scale shared-memory machines. From Pentium quads used in hardware systems, such as the Sequent NUMA-Q [51] or the HAL S-1 [80], to dual SPARC modules used in hybrid DSM systems, such as Typhoon-0 [63][72], and to four-processor AlphaServers used in all-software DSM systems, such as Shasta [69], all SMP nodes consist of two or four processors connected by a bus or a crossbar to a uniform access memory. In light of this trend, it is important to understand the interaction between SC-COMA's approach to coherence and the hardware provisions of an SMP platform and to evaluate the potential performance advantages for such an implementation.

6.1.1 SMP Clusters

SMP nodes have several significant advantages. Foremost is their appealing cost-performance ratio. By sharing memory and all the resources found in a conventional workstation between multiple processors, the cost per MIPS is much reduced. As processors already include support for snoopy coherence protocols, there is very little additional cost to connect several processors together.

There are several reasons why the sharing of attraction memory between the processors in an SMP node can improve the performance of a COMA:

- Lower node capacity miss ratio. Replicated data use fewer attraction memory frames per processor, thus the pressure on the attraction memory is smaller.
- Fewer replacements. There are two causes for this. First, for the same reasons the capacity miss ratio is smaller, there are fewer victimizations in the attraction memory. Second, when a replacement is sent out, it will travel less on average until accepted by another node simply because there are fewer nodes to visit.
- Lower node coherence miss ratio. Some of the applications' inherent communication between co-located processors is accomplished by means of the low-latency intra-cluster medium. Secondly, some of the coherence misses occurring in uniprocessors are eliminated in SMP nodes by prefetching effects. This is especially true for one-to-many or all-to-all data communication patterns. Prefetching effects also reduce the amount of cold misses at the node level.
- There is less request forwarding. The chances that an attraction memory miss happens at the home node or that the current owner is the home node increase by the factor of clustering (assuming a random distribution of data to home nodes).

Additional to the above performance advantages, a software-implemented coherence protocol can take advantage of an SMP node to optimize and balance the load due to the protocol activity. Optimizations can include the scheduling of external request processing on processors which are already idle due to remote misses or synchronization. Load balancing can reduce the average queueing time for messages, thus improving the remote latency. At the protocol level, there are fewer invalidations to send out and to process, which is a significant advantage for a software implementation when dealing with widely shared data.

The implementation of a software protocol between SMP nodes is not trivial though. Most of the problems are raised by the possibility of races when multiple processors are involved in protocol actions. There are also issues raised by the fact that protocol processing can be scheduled on more than one processor and, likely, some policies work better than others. At the same time, the level of hardware support for a software protocol should be kept at a minimum.

6.1.2 Implementation Issues and Solutions

The sharing of hardware resources between the processors in an SMP node and the interaction between multiple protocol activities scheduled simultaneously raise several hardware and software issues which are entirely new and will be analyzed next.

Hardware Support

The ACD must be able to record and report information about attraction memory misses to each processor in the node on an individual basis. A register file for miss status must be implemented in the ACD for this purpose. Also, the identity of the processor must be known to the ACD during a memory access. Processors can retrieve miss information from the register file by using their identifier as an index.

Cache invalidate and flush operations must be visible at the bus level and can no longer be implemented with special instructions which would control a single cache, that of the processor executing these instructions. Consequently, bus transactions for flushing and invalidating cache blocks must by issued by a special device under the control of the processors executing protocol activity. This device can be incorporated in the ACD.

Special care must be taken during the sequence of operations required for the downgrading of lines to avoid refetching of the data into the cache of another processor in the node. The software solution to this is to first lock the data in the attraction memory by using a special state. However, the ACD must be prepared to handle write-backs and other special operations to data in locked state.

To improve the memory bandwidth, SMP nodes are commonly provided with interleaved memory modules. The ACD must perform two memory accesses for every shared memory reference: one for the tags and another for the data. We assume that the memory is two-way interleaved and that attraction memory data and the associated tags can always be fetched in parallel (the CAS cycle for data begins only after the tag comparison is done and the full memory address for the data is available).

There are many new features in the network interface that are desirable in the context of SMP clusters. Sharing of the outbound network interface between processors must be made free of interference. The dialog between any processor and the network interface can be encapsulated in a critical section, but there could be performance gains from providing individual control and status registers for each processor. For the inbound messages, a smart dispatcher of requests and replies to the appropriate processors might help balance the load and reduce the software overhead. In the absence of such a dispatcher, messages can be routed to processors statically, based on the line address in their header, or randomly.

Software Modifications

SC-COMA uses a shared directory at the level of the SMP node. In the event of a miss, every processor in the node executes its own access fault handler and issues the appropriate remote request. To prevent situations when two processors miss on the same line and would issue the same request, there must be a locking mechanism. Such locks must be added for every attraction memory set because the opportunities for interference extend beyond the line level, as in the situation presented above. Other scenarios include the victimization or invalidation of a shared line which has a recent ownership miss and the acceptance of a replacement right after the same line has generated a miss on another processor. In general, these are cases which make the attraction memory state information, as seen and recorded by the ACD when the access was faulted, to become obsolete by the time the miss handler retrieves this information from the ACD registers. The solution is to read the attraction memory state only after acquiring the lock.

Due to SC-COMA's method of completing an access after the missed data was obtained (i.e. by copying the missing instruction and re-executing it from a special location in the code of the miss handler), the kernel must maintain separate physical pages for the access fault handlers for each processor in the node. Also, closing the window of vulnerability becomes more complex, as the missed data may be removed by the actions of another processor (either on behalf of another node or just by victimization). To avoid this problem, fresh data must be loaded in the attraction memory in a special state until the miss is completed.

6.1.3 Performance Evaluation

To evaluate the performance of SC-COMA on SMP clusters we compare 32-processor systems. Thus, the uniprocessor version of SC-COMA has 32 nodes. The SMP version uses eight four-way SMP nodes. The processors in an SMP node are connected by a split-

phase bus running at 100MHz. A snoopy protocol (MESI) maintains hardware coherence at the node level. Main memory in the SMP nodes is two-way interleaved. The total amount of attraction memory is kept constant between the two systems (i.e. the SMP nodes have four times larger memories than the uniprocessor nodes) and the memory pressure is 75%.

Table 6.1 compares the node miss ratio and the replacement rate of six benchmarks executing on the above mentioned implementations of SC-COMA. With the exception of FFT, all applications exhibit significantly lower node miss ratios.

Benchmark	Node Miss Ratio (%)		Replacement Rate (per 10^6 references)	
	32 uniprocessors	8 four-way SMPs	32 uniprocessors	8 four-way SMPs
Barnes	0.158	0.057	95.0	59.9
FFT	0.467	0.454	343.8	934.3
LU	0.081	0.035	182.9	56.7
Ocean	0.519	0.193	1522.8	943.9
Radix	1.518	0.865	6165.8	4355.9
Raytrace	0.781	0.411	273.8	460.2

Table 6.1. Node miss and replacement ratios for uniprocessors and SMP clusters
32-processor systems. 75% memory pressure, relaxed inclusion, replacement hints.

Both Barnes and Raytrace see a sharp reduction in the amount of capacity misses and coherence misses, as well, in the SMP version. This is due to the fact that the degree of data replication is large in both applications and clustering takes advantage of this.

In FFT, the SMP version has no opportunity to save capacity misses because there are already very few in the uniprocessor SC-COMA. Also, the only reduction of coherence misses occurs during an all-to-all communication phase (the matrix transposition) when there is a 3/32 higher chance that communication is local. The reason the replacement rate is higher in the SMP version is due to the sequential phase. As the attraction

memory size per node quadruples, processor 0 is able to accumulate more data while performing initialization. Much of this data incurs replacements when the parallel section begins.

Ocean (which is the "contiguous" version of the code) has predominantly producer-consumer communication between neighboring nodes on a grid. Coherence misses are eliminated when some of these neighbors are located in the same SMP node.

LU has little intra-cluster communication, but clustering helps reduce the capacity misses, especially for the widely shared data. This is confirmed by the much lower rate of replacements.

The write capacity miss ratio, which makes up the bulk of the node miss ratio, is significantly lower for Radix in the SMP. There is also a small decrease in the coherence miss ratio during an all-to-all communication phase.

Figure 6.1 compares the overall execution times for six benchmarks on two systems: the base SC-COMA (with 32 uniprocessor nodes) and the implementation of SC-COMA using SMP clusters (eight four-processor nodes). As can be seen, the performance advantages of using SMP nodes are clear. In some cases, such as LU and Barnes, the SMP SC-COMA performs almost as well as the previously analyzed hardware-coherent HW-COMA (see Figure 4.3).

The execution times are broken down into five categories. The busy time contains all the time spent in executing the application without any stalls due to synchronization (synchronization time) or memory. Local memory access stall includes all the delays due to secondary cache hits, cache-to-cache transfers (for the SMP version), and local attraction memory hits. The remote stall includes all the delays for accesses that must be completed by taking software actions to bring data from remote nodes or to gain exclusivity. Finally, some of the execution time is due to the processor being used for protocol activities when the application is not stalled otherwise.

In addition to the expected decrease in the amount of remote stall and interruption overhead in the SMP version, as predicted by the node miss ratios from Table 6.1, there are other effects. The local stall increases with respect to the base system due to two reasons. First, more accesses are completed locally, either by cache-to-cache transfers or because of the higher attraction memory hit ratio. Secondly, attraction memory hits tend to get slower, as processor cache misses must contend for the cluster bus and the main memory. Applications with higher processor cache miss ratios are more affected by this last reason: Ocean, Radix, and Raytrace. Finally, there is an overall decrease in the amount of synchronization delays, as the execution of critical paths becomes faster.

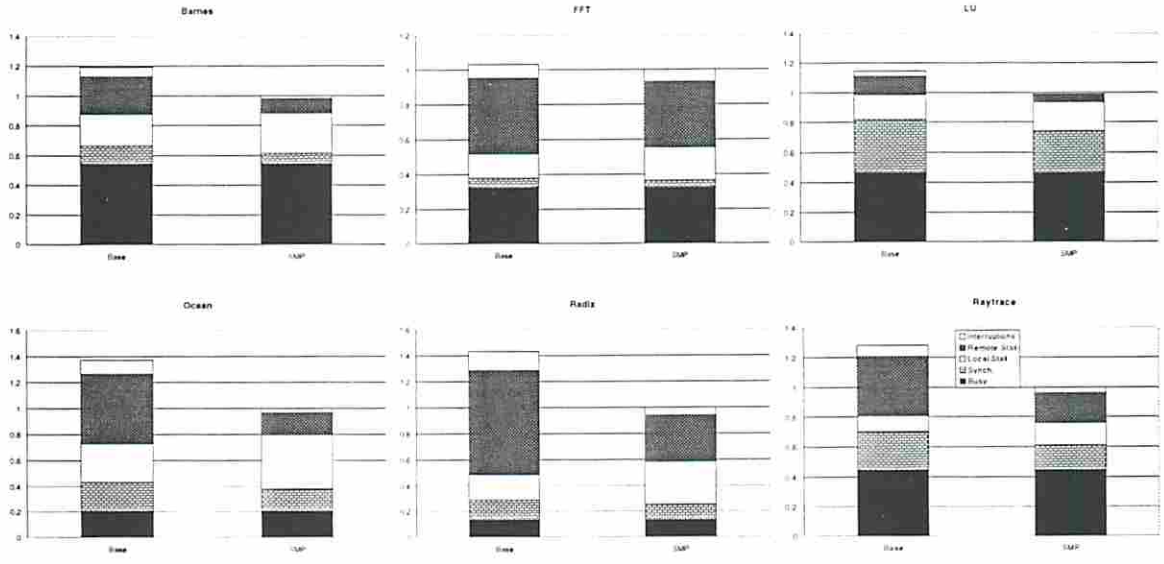


Figure 6.1. Execution times for the base and the SMP SC-COMA

75% memory pressure, relaxed inclusion, replacement hints.
The execution times are normalized with respect to the SMP SC-COMA.

6.2 Incorporating Non-Blocking Stores in SC-COMA

In this section we are examining the feasibility and the performance consequences of relaxing our previous assumption that a processor blocks during store operations. Because stores only propagate values to the memory and, typically, do not affect the processing of instructions immediately following in the execution stream, there is significant gain from the overlap of stores with computation. All modern processors are provided with store buffers, which support the decoupling of stores from the execution pipeline and handle the stores autonomously until their completion.

There are two levels at which non-blocking stores can be incorporated in SC-COMA. At the first level, local store hits are non-blocking for the processor by virtue of the store buffer. At the second level, store misses are non-blocking for the protocol engine, i.e. after a write miss or ownership request has been issued after a store miss, the interrupted application thread is resumed. There is considerably more software complexity in supporting this second level and, likely, there are some performance advantages as well. However, there is nothing additional to the first level in terms of required mechanisms and recovery techniques. Thus, we will concentrate on the first level and assume that every local miss, load or store, triggers a protocol action which returns to the application only when the miss is completed. We should note that, due to the low write miss ratios in the large attraction memory for the SPLASH-2 benchmarks (except for Radix), there is less performance penalty for this decision than one might expect.

In the following, we describe store buffers and the problems they pose to software coherence executed on the main processor. The mechanisms and techniques by which a decoupled store can be brought to completion in SC-COMA after being faulted at the ACD level are presented next. Finally, we present results from the evaluations of SC-

COMA supporting non-blocking stores and a hardware implementation using an aggressive release consistency memory model [29].

6.2.1 Store Buffers

Stores decoupled from the processor's reference stream are deposited in the store buffer, as shown in Figure 6.2, where they will wait for completion at some level of the cache-memory hierarchy. Load references are allowed to bypass such buffered stores.

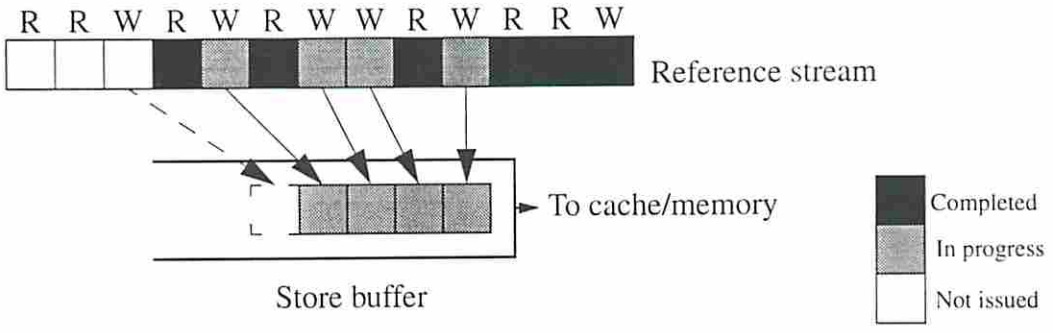


Figure 6.2. The store buffer

Processors with on-chip first-level caches (FLC) usually include a first-level store buffer as well (see Figure 6.3). The FSB hides the latency of writing through the FLC into the second-level cache (SLC), which is external to the processor. The processor stalls on a write only when the FSB is full. Stores buffered in the FSB which hit in the SLC are considered completed, but the ones that miss are usually transferred to a Secondary Store Buffer (SSB) between the SLC and memory. This helps to reduce the chances that the FSB fills up. In NUMA, the memory actually consists of local and remote memory and, for performance reasons, it is important to allow an SLC miss to complete in the local memory (or to trigger a remote access) while a previous SLC miss is still pending. Essentially, the SLC must allow multiple outstanding misses, thus being a lockup-free cache. SC-COMA,

however, does not require lockup-free caches; an SLC miss either completes in the attraction memory or it is faulted, at which point all outstanding references (i.e. writes, because loads are blocking) become frozen[1]. In some sense, the second level of incorporation of non-blocking stores in SC-COMA amounts to the implementation of a software SSB for the main memory cache.

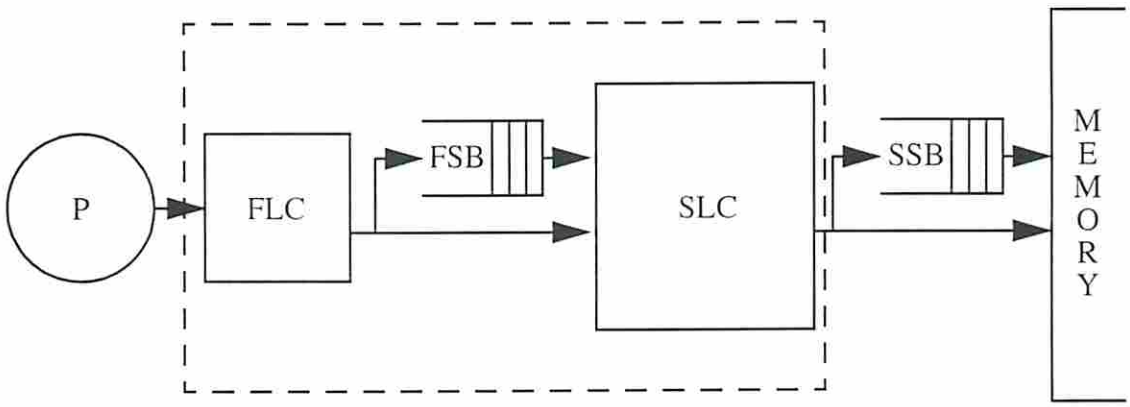


Figure 6.3. Store buffers between the memory hierarchy levels

In order to simplify the simulation and because processor stalls due to a full FSB are very small in practice, we will assume that there is a single level of write-back cache (shown with dotted line in Figure 6.3) and a store buffer between this cache and memory.

By the time a non-blocking store reaches the memory and, possibly, a miss is detected by the ACD, the processor program counter has advanced past the instruction that generated the store and might even have retired several other instructions following that store instruction. Consequently, for faulted stores, it is unfeasible to roll-back the processor status to the particular store instruction[2]. Rather, faulted stores are reported asynchro-

1. We are still assuming that the attraction memory is not pipelined. Otherwise, the ACD must be prepared to record multiple faults and the software coherence layer must handle such events.

nously, much like hardware interrupts. In conventional systems, such asynchronous access faults are typically viewed by the operating system as catastrophic errors. However, the store buffer usually includes provisions to record information (physical address, data, and status) about such a fault and to allow the faulted store to be safely retried at a later time[1]. The particular method of accessing the store buffer information may vary from one manufacturer to another (even for the same processor), as it is a diagnostic-mode procedure. The future discussion is based on specifications for the Texas Instruments SuperSPARC processor [76]. After retrieving the information about the faulted store, the access to the physical location can be retried by using special operations which bypass the MMU. In the Sun-4M architecture, special alternate-space store instructions (*sta*) are available for this purpose.

There are several issues regarding the interaction of software coherence with the store buffers. Of main concern is the status of writes pending in the write buffer when either the ACD or the network interface signal the processor that a coherence action must be performed. In particular, we will analyze what happens when one of the following three events occurs: an asynchronous interrupt from the network interface, a synchronous trap after a load is faulted by the ACD, and an asynchronous trap after a buffered store is faulted by the ACD. Other issues concern performance aspects, such as the optional use of store buffers during protocol processing or the consequences of software coherence on the store buffer size. These issues are discussed next.

---

2. Often times, it is impossible to even link a faulted store to a particular instruction.

1. This information can also be recorded by the ACD.

6.2.2 Incorporating Store Buffers in SC-COMA

Several hardware mechanisms allow SC-COMA to enable the use of store buffers and to safely execute applications and coherence actions on the same processor. These mechanisms are:

1. As soon as a store buffer exception is detected, the store buffer is frozen: the store buffer is disabled and all the pending writes, including the faulted one, are retained in the buffer.
2. Store buffer exceptions are reported to the CPU via a data store error trap.
3. A faulted store can be retried by simply re-enabling the store buffer after the trap is handled. When a bypass operation is used to complete the store, the store buffer pointers can be adjusted to skip the faulted store when the buffer is re-enabled.
4. Data store error traps can be masked off. When the traps are disabled, store buffer exceptions can be detected by polling store buffer status registers.
5. Upon an interrupt or a trap, the store buffer is drained before the interrupt/trap handler is started. If store buffer exceptions occur while the buffer is drained, a data store error trap has precedence over the interrupt/trap.
6. Data store exceptions have the highest trap priority.

When a data store trap occurs, there can be no outstanding load operations because the loads are blocking. Also, because the store buffer is frozen and maintained like a FIFO, no other data store exceptions can occur until the store buffer is re-enabled. After gathering the needed information, the data store exception handler in SC-COMA is ready to take action to complete the store by issuing a remote request.

There are questions of whether and when to re-enable the store buffer and how to handle the other pending stores in the buffer (some of which may potentially miss as well). A very simple solution is to keep the buffer disabled, thus forcing all the writes to be syn-

chronous while the trap handler is running. This way, all the pending stores are kept intact in the buffer. When the coherence actions are done and the trap handler is just about to terminate, the store buffer is re-enabled. The previously faulted store will hit this time[1] and complete, while the other pending stores will be handled as if no exception ever took place. There is little performance loss because the store buffer is not re-enabled during the trap because of two reasons. The amount of write misses in the attraction memory is usually very small and the number of writes in the code of the trap handler is also reduced. It is true that some of the handler's writes are non-cacheable, but all efforts are done to keep them, and data updates in general, away from the critical miss path (such as updating the attraction memory's Tag Table or the directory after the request was issued). Commands sent to the network interface/DMA are, however, on the critical path and they require non-cacheable stores.

A second solution is to re-enable the store buffer in the trap handler soon after a request was issued (thus, during otherwise idle time). Before enabling the buffer, the faulted store must be removed from the buffer. This store must be recreated later using a bypass operation. With data store exceptions masked off, the store buffer will drain under software supervision for other potential exceptions. Polling is used for this purpose. If the buffer is seen to become empty (the fill and drain pointers are equal) and if no exceptions are flagged, the re-enabling was completed successfully. In the event of another exception, to avoid any complications caused by the simultaneous handling of two write misses, the store buffer is not re-enabled. A fresh data store exception will be generated for the second miss after the trap handler for the first miss terminates. The likelihood of two attraction memory write misses in such a small window of time is very little and limited mostly to

1. Note that the window of vulnerability is closed because interrupts, which might remove the line, are not acknowledged until the store buffer is drained.

situations of one-time data initialization. Thus, there is a very small performance penalty for this decision.

When a network interrupt occurs, prior to acknowledging it and starting the handler for external requests, the store buffer is drained and potential store buffer exceptions are handled first. This feature both guarantees that an external request cannot be replied to with stale data (if a store is still pending in the buffer) and that no store buffer exceptions can occur in unexpected situations.

When a synchronous trap is caused by a missing load, prior to acknowledging it and starting the miss handler, the store buffer is drained and potential store buffer exceptions are handled first. If a store buffer exception does occur during the draining, the load is restarted when the store buffer exception handler returns and will again incur a synchronous trap (unless the previous write miss has brought in the data).

Stalls due to full store buffers are less of a concern for SC-COMA than they are for HW-COMA, as only the local memory hierarchy is visible at the hardware level. In general, SC-COMA is awkward at dealing with multiple levels of store buffers but, fortunately, it does not need a store buffer for secondary cache misses. However, in order to take full advantage of the possibility to overlap writes with computation, SC-COMA would have to manage multiple write misses in software. This complicates the coherence handlers and possibly increases the latency of read misses. Fortunately, due to COMA's large attraction memory, most of the remote write misses are due to coherence (true or false sharing) and not capacity, thus they are expected to be relatively small for many applications.

It is relatively straightforward to allow a single outstanding write miss in SC-COMA and only block in the protocol engine on the next read or write miss. This optimization, similar to optimizations of sequential consistency [1], could bring further improvements because, again, attraction memory misses are reasonably spaced apart. On the other

side, the latency of a miss is higher as well. Although not implemented, we will evaluate the applicability of this optimization by examining the distribution of the time interval between a remote write/ownership miss and a subsequent remote miss of any kind.

6.2.3 Performance Evaluation

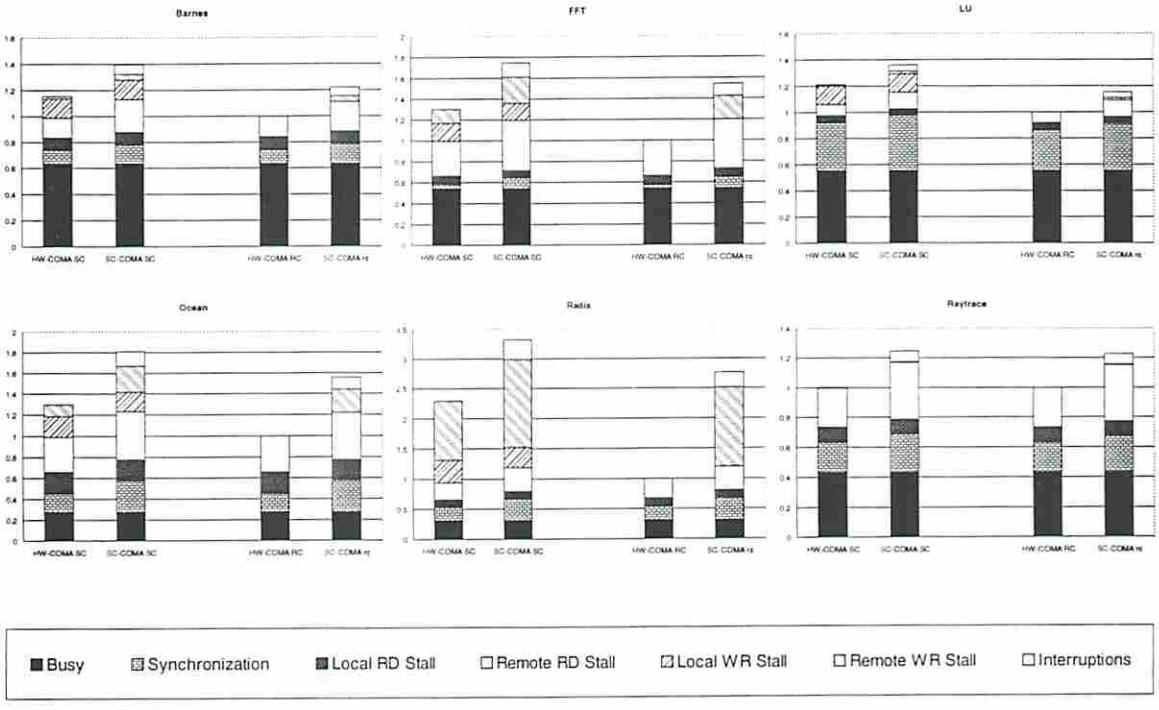


Figure 6.4. Effects of store buffers on the execution time

32 node systems, 75% memory pressure, relaxed inclusion. The two bars on the left correspond to a sequential consistency memory model. On the right, HW-COMA uses release consistency with unlimited number of outstanding write misses, while SC-COMA only takes advantage of store buffers for local write hits. The execution times are normalized with respect to the release-consistent HW-COMA.

Figure 6.4 shows the execution times of six benchmarks for four systems: the sequentially-consistent and the release-consistent HW-COMA and SC-COMA. The release-consistent SC-COMA is less aggressive then its hardware counterpart because the processor blocks on every remote write/ownership miss. As can be seen, the general trend is that HW-COMA increases its performance advantage under release consistency. How-

ever, with the exception of Radix, SC-COMA is still competitive. By eliminating the local write stall in both HW-COMA and SC-COMA, as well as the remote write stall in HW-COMA, release consistency is a significant improvement over sequential consistency. With the exception of Raytrace, which has a minute amount of writes to shared data, all the benchmarks show execution times which are 15-55% smaller when writes are non-blocking.

There are several common effects. Relaxed consistency leads to a compression of the execution time. This creates more contention in the network and slightly higher average remote latencies. For HW-COMA, time compression also creates more contention at the memory and higher local read stalls. In SC-COMA, the path between the processor and memory is not arbitrated and the local read stall is unaffected. Of course, time compression in SC-COMA creates higher queuing times for messages waiting to be scheduled on the protocol engine, thus increased remote latencies. However, the use of store buffers reduces the software overhead during protocol processing, thus counterbalancing the queuing time. The reduced software overhead is visible in the "interruptions" category as well. Finally, the synchronization penalty can be favorably influenced by the smaller write stalls, as load imbalances tend to become less prominent, best seen in LU.

Benchmark	Node read miss ratio (%)	Node write miss ratio (%)	Node ownership miss ratio (%)
Barnes	0.154	0.004	0.032
FFT	0.392	0.074	0.169
LU	0.077	0.004	0.016
Ocean	0.437	0.082	0.289
Radix	0.365	1.153	0.027
Raytrace	0.781	0.000	0.009

Table 6.2. Node miss ratios to shared data for the SPLASH-2 benchmarks
Percentages of all references for 32-processor systems, 75% memory pressure, relaxed inclusion, replacement hints.

To further discuss the results for each application in part, Table 6.2 indicates the node ratios for write miss and ownership transactions. For comparison purposes, the node miss ratio for reads is listed as well.

Barnes has very small node write and ownership miss ratios. This makes the limitations of SC-COMA in overlapping write misses with computation become less apparent. Similarly, Raytrace, with an even smaller write miss ratio, would see no further improvements from an aggressive release consistency.

FFT, LU, and Ocean are single-writer applications and they should have almost exclusively ownership misses. The very small write miss ratios from Table 6.2 are actually due to capacity misses at the 75% memory pressure. Even so, the amount of remote write stall is still negligible. Ocean and FFT would likely see improvements on SC-COMA if ownership misses are non-blocking, but the stall due to an ownership miss is usually smaller than a read/write miss stall (unless data is widely shared and many invalidation acknowledgments have to be processed).

The very high node write miss ratio explains why the performance of Radix on SC-COMA is so bad, compared to HW-COMA. Although part of the node write misses from Table 6.2 are due to capacity, the cold and coherence write misses are very high in this application and there are no hopes that things might get better if the memory pressure is lowered. The only way to improve the performance of Radix is by using a more aggressive implementation of relaxed consistency.

We now examine the potential performance improvements in SC-COMA by allowing the protocol engine to return to the application after issuing a write/ownership request instead of blocking. The protocol engine still blocks, and waits for the pending write to finish, when the next miss occurs. If the distance in cycles between a write miss and the next miss is large enough, it is possible that a good part of the (or the entire) write latency was successfully hidden. As an indicator, we will use the cumulative distribution of this

distance, as opposed to the average distance [30]. In some cases, a high average distance would falsely indicate that the write latency is hidden in a high proportion when, in fact, the average is high just because of a small percentage of large distances. To be able to extrapolate our conclusions to multiple-issue processors, we will also show distributions of the distance measured in the number of instructions.

Figure 6.5 shows the cumulative distribution of the distance between a write miss in the attraction memory and the next miss. The data was collected for HW-COMA and the distance is measured in cycles. As can be seen, in four applications (Barnes, LU, Ocean, and Raytrace) there is significant opportunity to hide the latency of write misses in SC-COMA even if just one write miss can be outstanding at any time. In general, a distance of 1000 cycles is enough to assume that a pending write miss has completed and a distance of 500 cycles is enough for an ownership miss with just one invalidation. For example, in Barnes, it is reasonable to assume that 30% of write misses will be completed by the time the next miss occurs, thus, their latency is totally hidden. Similarly, for Ocean, about 45% of ownership misses should complete before another miss occurs.

It is hard to say which of the two applications, Radix and FFT, offers less opportunity to hide the latency of a write miss. FFT has mostly ownership misses (hence the window of completion is about 500 cycles), whereas Radix has predominantly write misses (and the window is 1000 cycles). Nevertheless, at least 15-20% of the remote write stall can be still removed by allowing a single outstanding write miss in SC-COMA.

Figure 6.6 shows the same distance as Figure 6.5, but measured in instructions. This data is dependent on application characteristics and memory pressure (which affects the node hit ratio) only. Multiple-issue processors have an even shorter window of computation time to overlap with the latency of a previous write miss. For example, a four-issue processor executing FFT will issue instructions only 11 times before a new remote miss follows a write miss in 80% of the cases.

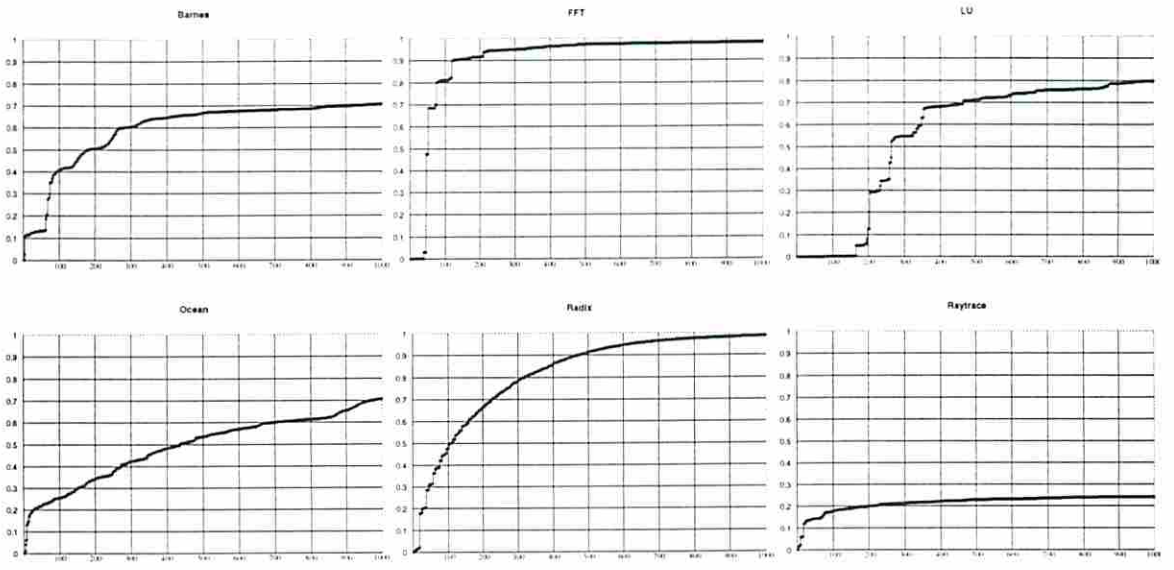


Figure 6.5. The distance between a write miss and next miss (cycles)

The graphs indicate the cumulative distribution of the distance measured in cycles for HW-COMA RC.

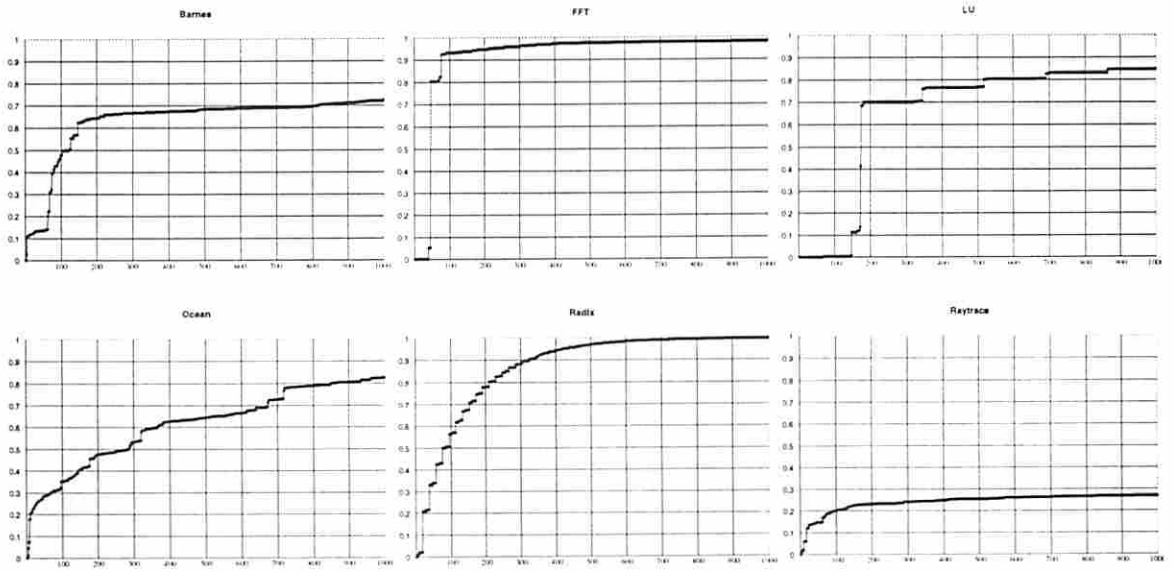


Figure 6.6. The distance between a write miss and next miss (instructions)

The graphs indicate the cumulative distribution of the distance for up to 1000 instructions.

As compared to HW-COMA, due to the higher latency (hence, the larger window of completion for a write miss), SC-COMA is less able to take advantage of the application-specific distance between remote misses and hide the write latency by allowing a single outstanding miss. Hence, aggressive relaxation of consistency might be required for some applications to achieve a high level of performance on SC-COMA. As mentioned, aggressive relaxation could increase the latency of reads. This is of little importance in Radix, for example, but might be harmful to the overall performance in other applications. Again, the flexibility of a software protocol allows application-specific customizations of the memory consistency model.

Chapter 7

RELATED WORK

There is a large body of related work concerning the performance evaluation and design considerations of hybrid DSM systems and other approaches to DSM by means of software protocols. There is also a significant amount of literature dedicated to many aspects of COMA architectures.

7.1 Hybrid DSM Systems

Software cache-coherence was first proposed for the management of processor caches in the VMP multiprocessor [15].

Alewife first prototyped a version of hybrid DSM. Software-extended protocols were used to build a CC-NUMA machine where the simple features of the protocol were supported by hardware and the infrequent, but complex situations, handled in software. The software protocol actions were written in C and executed on the main processor, which was equipped with support for fast context switching. A study has compared a software-only solution with a software-assisted solution and found it to be surprisingly good [12].

On the heels of the Alewife study, Grahn and Stenström evaluated various implementations of a NUMA protocol with software-only management of the directory on an NCC-NUMA substrate [33]. They employ the same idea of separating the state table, needed by the hardware to do access checking, from the directory data, used by the soft-

ware handlers to maintain presence information. A custom node controller provides support for remote put/get operations and implements a good part of the protocol actions. Software protocol actions are running on the main processor at the home nodes only. The design requires high-availability processor interrupts and lock-up free caches in order to prevent deadlocks, as well as an increased level of complexity in the node controller.

Typhoon [62] is one of the first proposals for hybrid DSM systems with high levels of integration. It is based on Tempest, a parallel machine interface for user-level shared memory. Typhoon's network device contains a processor dedicated to user-level protocol handling. Coherence events, snooped from the cluster bus or signaled by the network interface, invoke user-level procedures. Physical addresses from the bus are translated back into virtual addresses using a RTLB before being passed as arguments to the handlers. The memory organization is inspired from Simple COMA. Typhoon also provides hardware support for low-overhead messaging, bulk data transfers and virtual memory management. User-level protocol handling provides a maximum degree of flexibility at the cost of having to use a RTLB. This leads to slightly higher miss latencies than system-level handlers and increased costs and complexity. Also, for multiprocessor clusters, a single protocol processor may become a bottleneck. There are several working incarnations of Typhoon: Blizzard [71] on the CM-5 and Typhoon-0 [63] on a cluster of workstations.

The hardware support for fine-grain sharing in Blizzard-E [71] is based on a combination of ECC bits in the memory and page access right bits in the MMU to detect block access faults. Because only one bit is available in the ECC (encoding valid/invalid states), write operations usually take an expensive page fault, due to the page being mapped read-only, and resolve the access by checking extra tables for write permission. Thus, the latency of write hits is considerable. Blizzard multiplexes protocol and application on the same processor and it uses a low-overhead method of switching the processor between application and protocol processing, similar to SC-COMA. However, because of insuffi-

cient privileges at the user level, it must find ways around several idiosyncrasies of the operating system, with some impact on performance [64]. Blizzard's provision for guaranteeing forward progress is unclear. Faulting instructions are re-executed after returning from the exception, which leaves open a window of vulnerability. Efficient communication is supported by CM-5's user-level network interface.

Typhoon-0 [63] implements another Simple COMA on a cluster of SparcStation-20s. A custom MBus device, the Vortex [61], supports access control by using a state table stored in dedicated SRAM. The protocol handlers for Typhoon-0 are designed to run at user-level, offering it additional flexibility. One of the two processors in the workstation serves as a dedicated processor which runs protocols at the user level and detects coherence events through polling of the status registers in the Vortex. While the small overhead of multiplexing the main processor is eliminated, this could be a waste of valuable resources in a small-scale SMP.

START-NG [16] includes an Address Capture Device (ACD) to support fine-grain sharing in a manner similar to Typhoon-0. But, like Typhoon's incarnations, START-NG has no associative memory. Instead, a level-3 cache is managed in software, which puts a heavier penalty on all level-2 cache misses to both local and remote lines. The coherence protocol runs in user mode on one of the site's four processors. START-NG's dedicated protocol processor and its interaction with the ACD have strong similarities with the solution in Typhoon-0. A custom tightly-coupled network interface can be accessed by processors through their level-2 cache interface.

The Stanford FLASH [44] is the most aggressive proposal for a DSM with software-implemented coherence. The level of integration between the network interface, memory controller and I/O ports is very high. A custom node controller, the MAGIC chip, provides optimized, hardwired data paths between processor, memory, network and I/O. MAGIC also contains the processor implementing the coherence protocol along with its

own instruction and data caches. Currently, like SC-COMA, FLASH does not allow user-level handlers and runs protocol handlers at system-level. FLASH, however, provides support for low-overhead user-level message-passing. Although a COMA protocol is planned for evaluation, the only reports [36] available are for a NUMA protocol with a directory structure using dynamic pointer allocation. While the performance of FLASH is high, its cost is high as well. One MAGIC chip is required for every processor in the system, even if its utilization is low. By being a dedicated DSM, FLASH and its techniques will be inapplicable to commodity workstations.

7.2 Other Approaches to Software Coherence

Solving the data coherence problem by software means has been a topic of research for many years. Early proposals forced the programmer or the compiler to flush caches at synchronization points [14]. However, subsequent work has concentrated on system-level approaches.

Kai Li has pioneered Virtual Shared Memory (VSM) systems [49] for distributed systems. Research in this area has grown continuously, fueled by the emergence of networks of workstations. Since the first VSM, Ivy [50], which was using sequential consistency, many systems have proposed the use of relaxed consistency memory models to alleviate the problem of false sharing. Most systems today, of which Treadmarks [2] is the flagship, use aggressive implementations of release consistency which allow multiple writers and postpone communication until synchronization points. The performance tradeoffs between the use of such page-based release-consistent systems and hybrid systems with support for fine-grain sharing have been compared by Zhou et al. [86]. They find mixed results, but several aspects, such as operating system overhead, memory consumption, and support for multiple coherence granularity are ignored.

To alleviate problems with page-level sharing in software DSMs, some systems abandon the use of the virtual memory support for access checking and perform fine-grain access checking implemented entirely in software: Blizzard-S [71] and Shasta [68]. In the absence of adequate compilers, the application executable can be modified by preceding load/store instructions with a short code to perform the checking. While maintaining all the flexibility and portability of a software implementation, such systems incur an overhead for executing access checking of up to 35% in Shasta [68], additional to the protocol processing overhead. Furthermore, the consumption of memory can be significant because every page of shared data replicates to every processor using any part of the page for the entire lifetime of the application. Memory-informing operations [38] can help further reduce the access checking overhead in software-only DSM systems, but require special processor support, currently unavailable. This category of user-level software-only DSM has COMA-like features, by accumulating the working set in the local memory. However, allocation of data must still be done at the page grain, thus incurring all the fragmentation disadvantages of Simple COMA. Shasta has also explored many of the issues regarding the implementation on SMP nodes [69].

An interesting direction is the software emulation of CC-NUMA on top of an NCC-NUMA substrate. Following Petersen and Li's work [59] on virtual memory supported cache-coherence, Kontothanassis and Scott have proposed Cashemere [42]. This approach exploits the virtual memory system of the operating system. Because the granularity of sharing is at the page level and because coherence is maintained by flushing pages from caches, the applications must be written under relaxed consistency models. When a processor writes into a page, it appends the page to *weak lists*, which must be visited in software at the time of an *acquire* to flush the caches.

7.3 COMA Architectures

Research on the hardware implementations of COMA begun with the DDM [35], which was using hierarchical directories. As demonstrated by the KSR-1 [10], hierarchical directories increase the latency of remote accesses. This would become even worse with software-implemented directory management. The Flat COMA, COMA-F [40], was proposed to eliminate hierarchical directories. More research on the properties of COMA was done by the DICE group at University of Minnesota [39][47][46]. To our knowledge, there is no working prototype of a Flat COMA. The Illinois Aggressive COMA (I-ACOMA) [77] group is currently building one. There is also ongoing development of a bus-based COMA [18].

Simple COMA [67] is a proposed hardware DSM that achieves COMA-like fine-grain data replication and migration at the main memory level but avoids the tag checking overhead. Data replication and migration are accomplished by binding pages from a globally-shared space into the physical space of each node. Full associativity of the main memory cache is supported by the TLB in every node's MMU, thus eliminating the need to check tags on every memory access. Coherence is performed in hardware by associating state bits with every line and using an access control device to check permissions. Coherence transactions must carry the line identifier in the global space, requiring a translation from/to the physical address at the sender and at the receiver. In addition to eliminating tag checking, Simple COMA simplifies the replacement problem as well, because lines are always accepted at their home. The disadvantages of Simple COMA come from the page-level allocation approach: operating system overhead for page mapping/allocation, memory fragmentation, increased capacity misses and network traffic. Also, because the coherence protocol is supported in hardware, flexibility is limited.

Several studies have compared the performance of COMA to other systems under hardware implementations. The results of Stenström et al. [74] indicate that Flat COMA outperforms CC-NUMA for eight SPLASH benchmarks. Their study made several assumptions favoring COMA: very small processor caches, no tag-checking penalty, infinite attraction memories, and round-robin data placement. Still, the attractiveness of COMA is questioned because of the hardware complexity. Zhang and Torrellas compared COMA and RC-NUMA [85]. Using round-robin data placement for seven SPLASH-2 benchmarks, they show mixed results, with a 4% average slowdown for COMA. Saulsbury et al. [67] analyzed the performance of Simple COMA for three SPLASH benchmarks concluding that it performs comparably to COMA and CC-NUMA. The study, based on analytical modeling, was biased by several assumptions: relatively small data sets, a fairly large cache (64KB), and unrealistic costs for page faults and COMA replacements. More realistic studies [27][5] point out the same problems with Simple COMA that we have shown in this work.

Chapter 8

CONCLUSIONS

This research has explored the design issues and the performance characteristics of a COMA architecture in a hybrid DSM implementation, thus with a software implemented coherence protocol and hardware support for fine-grain access control. The implementation of a protocol in software has advantages in terms of cost, flexibility, and ease of development and upgrade over hardware implementations. It also enables the use of the relatively complex COMA coherence protocol, of which no hardware implementations exist to date. In turn, the COMA architecture helps to reduce the overhead caused by the software protocol by minimizing the node miss ratio. The fine-grain access control eliminates problems caused by false sharing and reduces the need to use very aggressive relaxations of the memory consistency model, thus supporting the efficient execution of all classes of applications.

SC-COMA is a hybrid COMA designed for platforms consisting of networks of workstations. The coherence protocol employed by SC-COMA has original features which optimize its performance under a software implementation by reducing the number of exchanged messages per transaction. One such feature is request buffering which eliminates the need to inform the home node about the completion of a write transaction. Also, SC-COMA uses a new directory structure where copysets migrate to the current owner. An original method is used in SC-COMA to re-execute an instruction which has incurred a miss in the attraction memory and guarantee that no livelock situations can occur.

A quantitative performance comparison between SC-COMA and its aggressive hardware counterpart HW-COMA was performed by means of execution-driven simulation for a subset of the SPLASH-2 benchmarks. The study was conducted assuming a sequential consistency memory model and at different memory pressures. The execution times are 11-56% slower on SC-COMA, mostly due to the increase in the remote access latency. The speedups of SC-COMA and HW-COMA were compared up to 32 nodes. A simple analytical model confirms the experimental results and explains why there are no reasons to fear saturation in SC-COMA for this type of benchmarks with moderately low miss ratios. A study of the effects of processor speeds on the performance slowdown of SC-COMA reveals that the gap between SC-COMA and HW-COMA shrinks at high speeds, but never drops below a threshold. An analytical model explains the threshold as a consequence of the unscalable component of the software overhead: uncacheable accesses. It also sheds light on the different behavior of applications when the processor speed is varied.

The performance of COMA was quantitatively compared to other hybrid architectures using different memory organizations. The study was conducted using different memory overheads and data placement strategies. As the most important factor of performance in the particular design space is found to be the node hit ratio, COMA outperforms RC-NUMA, CC-NUMA, and Simple COMA. The organization of fine-grain overflow caches in main memory (and not page caches as in Simple COMA) is crucial for achieving good performance, especially for reasonable memory overheads and for applications with irregular access patterns which do not allow good static data placements. As RC-NUMA and COMA require the same simple hardware support for organizing such main memory caches, we argue in favor of incorporating this support in the memory controller in a standard way.

SC-COMA's protocol and the flexibility of a software implementation enable three optimizations which are evaluated in part. First, inclusion relaxation is a valuable technique to reduce conflicts in the attraction memory caused by widely shared data, thus reducing the overall node miss ratio and the number of replacements. Second, replacement hints are effective in keeping the number of nodes visited by a replacement request as close to one as possible. Finally, mastership hints can help in some applications to transform three-hop transactions into two-hop ones.

Two studies have been done to extend the hybrid COMA architecture to other hardware platforms. The use of SMP clusters brings some challenges to the implementation of the coherence protocol, but also offers significant performance advantages. These advantages are the consequence of improved node hit ratio, due to sharing at the node level, and reduced replacement rates. By quantitative evaluation we find that, in some cases, the SMP SC-COMA performs just as well as a base HW-COMA. Secondly, we adapt SC-COMA for the use of processors with non-blocking writes. A quantitative comparison with the sequentially-consistent SC-COMA shows that the elimination of local write stalls is already an important improvement. By comparing with an aggressive release-consistent HW-COMA, we show that, in some cases, SC-COMA would lag behind if only one outstanding write miss is allowed by the coherence protocol. In other cases, however, there is no need for multiple outstanding write misses and, sometimes, even sequential consistency performs well.

8.1 Future Work

This work can be extended in several ways. First, and most importantly, is to find very efficient ways to integrate and execute the coherence protocol on the main processor. As processors approach the memory wall, execution units will idle for significant fractions of time. They can be put back to use by either a special on-chip instruction unit acting as a

protocol engine, or by dedicated coherence threads in a multi-threaded processor. The access fault detection mechanisms can be integrated more tightly with the processing pipeline to avoid late notifications from the main memory level. A possible solution is to unify the TLB check with the fine-grain access checking.

Second, there is a number of interesting features which the coherence protocol could support as application-specific customizations. Multiple coherence granularities can better suit the different data access patterns and spatial locality properties across, or even within, applications, thus reducing the necessary misses and the data traffic. Mixed protocols, with selectable invalidate or update-based coherence can also lead to better performance for some applications.

Third, hybrid architectures are a fertile field for experimenting with adaptive memory hierarchies. As mentioned in Section 4.3.6, an interesting idea is to combine tag-free main memory, for data with mostly coherence misses or read-only pages replicated under operating system control, with remote data caches organized in main memory, for replicated data incurring capacity misses, and with attraction memory, for migratory data.

Fourth, there are many operating system issues which are worthy of investigation. Methods of controlling the memory pressure based on feedback from the protocol engine (i.e. the miss ratio or replacement rate) and data placements policies can help optimize the overall system performance. I/O and virtual memory management are also important topics for a full implementation.

Fifth, the performance of SC-COMA could be compared with page migration and replication controlled by the operating system, or with that of shared virtual memory systems, such as Treadmarks.

Finally, the results presented here could be verified for larger systems and for a wider variety of applications, of particular interest being commercial applications. As the

simulation technology is advancing quite fast and the computing power gets bigger and faster, it is without doubt that such studies will be possible in short time.

Bibliography

[1] S.V. Adve and M.D. Hill. Implementing Sequential Consistency in Cache-Based Systems. In *Proc. of the 1990 Int'l Conference on Parallel Processing (ICPP'90)*, pages 47–50, August 1990.

[2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54-64, February 1995.

[4] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. In ACM Transactions on Computer Systems 4(4): 273-298, November 1986.

[5] S. Basu and J. Torrellas. Enhancing Memory Use in Simple COMA: Multiplexed Simple COMA. In *Proc. of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[6] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proc. of the 17th Annual Int'l Symposium on Computer Architecture (ISCA'90)*, pages 125–135, May 1990.

[7] M. Björkman, F. Dahlgren, P. Stenström. Using Hints to Reduce the Read Miss Penalty for Flat COMA Protocols. *Proc of the 28th Hawaii Int'l Conference on System Sciences*, pages 242-251, January 1995.

[8] M. Blumrich et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. *Proc. of the 21st Annual Int'l Symposium on Computer Architecture*, pages 142-153, April 1994.

[9] W.J. Bolosky. Software Coherence in Multiprocessor Memory Systems. PhD. Thesis, Department of Computer Science, University of Rochester, 1993.

[10] H. Burkhardt III et al. Overview of the KSR-1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[11] L.Censier and P.Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[12] D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. *Proc. of the 21st Annual Int'l Symposium on Computer Architecture*, pages 314-324, April 1994.

[13] R.Chandra, K.Gharachorloo, V.Soundararajan, and A.Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *Proc. of the 8th ACM Int'l Conference on Supercomputing*, pages 274–288, July 1994.

[14] H. Cheong and A.V. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *IEEE Computer* 23(6):39-47, June 1990.

[15] D.R. Cheriton, G.A. Slavenburg, and P.D. Boyle. Software-Controlled Caches in the VMP Multiprocessor. *Proc. of the 13th Annual Int'l Symposium on Computer Architecture*, pages 366–374, June 1986.

[16] D. Chiou et al. StarT-NG: Delivering Seamless Parallel Computing. *Proc. of the First Int'l Euro-Par Conference*, pages 101-116, August 1995.

[17] A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwanpoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. *Proc. of the 21st Annual Int'l Symposium on Computer Architecture*, pages 106-117, April 1994.

[18] F. Dahlgren and A. Landin. Reducing the Replacement Overhead in Bus-Based COMA Multiprocessors. In *Proc. of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA-3)*, February 1997.

[19] M. Dubois, C. Scheurich, and F.A. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. of the 13th Annual Int'l Symposium on Computer Architecture (ISCA'86)*, pages 434–442, June 1986.

[20] M. Dubois, J. Skeppstedt, and P. Stenström. Essential Misses and Data Traffic in Coherence Protocols. *Journal of Parallel and Distributed Computing*, (29)2:108-125, September 1995.

[21] S. Dwarkadas, P. Keheler, A.L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. *Proc. of the 20th Annual Int'l Symposium on Computer Architecture*, pages 144-155, May 1993.

[22] S.J. Eggers and R.H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, April 1989.

[23] A. Erlichson, B.A. Nayfeh, K. Olukotun, and J.P. Singh. The Benefits of Clustering in Shared-Address-Space Multiprocessors: An Applications-Driven Investigation. In *Proc. of Supercomputing'95*, December 1995.

[24] A. Erlichson, N. Nuckolls, G. Chesson, and J.L. Hennessy. SoftFlash: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proc. of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 210–220, October 1996.

[25] B. Falsafi, A.R. Lebeck, S.K. Reinhardt, I. Schoinas, M.D. Hill, J.R. Larus, A. Rogers, and D.A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proc. of Supercomputing'94*, pages 380–389, November 1994.

[26] B. Falsafi and D. A. Wood. Scheduling Communication on an SMP Node Parallel Machine. In *Proc. of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA-3)*, February 1997.

[27] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proc. of the 24th Annual Int'l Symposium on Computer Architecture*, pages 229-240. June 1997.

[28] B. Falsafi and D. A. Wood. When does Dedicated Protocol Processing Make Sense? Technical Report 1302, Computer Sciences Department, University of Wisconsin-Madison, February 1996.

[29] K. Gharachorloo, D.E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th Annual Int'l Symposium on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[30] K. Gharachorloo, A. Gupta, and J.L. Hennessy. Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors. In *Proc. of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 245–257, April 1991.

[31] A. Gefflaut, A. Moga, J. Jeong, and M. Dubois. Design and Evaluation of a Software-Controlled COMA. Technical Report 96-03, EE-Systems, University of Southern California, January 1996.

[32] R.B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[33] H. Grahn and P. Stenström. Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors. *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture*. pages 38-47, June 1995.

[34] A. Gupta, W.-D. Weber: Cache Invalidation Patterns in Shared-Memory Multiprocessors. In *IEEE Transactions on Computers* 41(7): 794-810, 1992.

[35] E. Hagersten, A. Landin, and S. Haridi. DDM--A Cache-Only Memory Architecture. *IEEE Computer*, (25):9, pages 44-54, September 1992.

[36] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. *Proc. of the Sixth Int'l Conference on Arch. Support for Programming Languages and Operating Systems*, pages 274-285, 1994.

[37] C. Holt, M. Heinrich, J.P. Singh, E. Rothberg, J. Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[38] M. Horowitz, M. Martonosi, T.C. Mowry, and M.D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. *Proc. of the 23rd Annual Int'l Symposium on Computer Architecture*, pages 260-270, May 1996.

[39] S. Jamil and G. Lee. Unallocated Memory Space in COMA Multiprocessors. In *Proc. of the 8th Int'l Conference on Parallel and Distributed Computer Systems*, September 1995.

[40] T. Joe. COMA-F: A Non-Hierarchical Cache Only Memory Architecture. PhD. Thesis, Stanford University, March 1995.

[41] J. Kong and G. Lee. Relaxing the Inclusion Property in Cache Only Memory Architecture. In *Proc. of the Second Int'l Euro-Par Conference*, volume II, pages 435–444, August 1996.

[42] L.I. Kontothanassis, M.L. Scott. Software Cache Coherence for Large Scale Multiprocessors. *Proc. of the 1st Symposium on High-Performance Computer Architecture*, pages 286-295. January 1995.

[43] J. Kubiatowicz, D. Chaiken and A. Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. *Proc. of the 5th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274-284, October 1992.

[44] J. Kuskin et al. The Stanford FLASH Multiprocessor. *Proc. of the 21st Annual Int'l Symposium on Computer Architecture,* pages 302-313, April 1994.

[45] A. Landin and M. Karlgren. A Study of the Efficiency of Shared Attraction Memories in Cluster-Based COMA Multiprocessors. *Proc. of the 11th Int'l Parallel Processing Symposium*, April 1997.

[46] G. Lee. An Assessment of COMA Multiprocessors. In *Proc. of the 9th Int'l Parallel Processing Symposium (IPPS'95)*, pages 388–392, April 1995.

[47] G. Lee and S. Jamil. Memory Block Relocation in Cache-Only Memory Multiprocessors. In *Proc. of the Seventh IASTED/ISMM Int'l Conference on Parallel and Distributed Computing and Systems*, October 1995.

[48] D.E. Lenoski et al. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. *Proc. of the 17th Annual Int'l Symposium on Computer Architecture*, pages 148-159, 1990.

[49] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[50] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. *Proc. of the 1988 Int'l Conference on Parallel Processing,* volume II, pages 94-101, August 1988.

[51] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proc. of the 23rd Annual Int'l Symposium on Computer Architecture*, pages 308-317. May 1996.

[52] M. Marchetti, L. Kontothanassis, R. Bianchini, and M.L. Scott. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *Proc. of the 9th Int'l Parallel Processing Symposium*. April 1995.

[53] M.M. Michael, A.K. Nanda, B-H. Lim, and M.L. Scott. Coherence Controller Architectures for SMP-based CC-NUMA Multiprocessors. In *Proc. of the 24th Annual Int'l Symposium on Computer Architecture (ISCA'97)*, pages 219–228, June 1997.

[54] A. Moga, A. Gefflaut, and M. Dubois. Memory Organizations in Hybrid DSM: A Performance Comparison. Technical Report 97-02, Computer Engineering Department, University of Southern California, February 1997.

[55] A. Moga, A. Gefflaut, and M. Dubois. Hardware versus Software Implementation of COMA. *Proc. of the 1997 Int'l Conference on Parallel Processing,* pages 248-256, August 1997.

[56] H.L. Muller, P.W.A. Stallard, and D.H.D. Warren. The Application of Skewed-Associative Memories to Cache Only Memory Architectures. *Proc. of the 1995 Int'l Conference on Parallel Processing*, vol. I, pages 150-154, 1995.

[57] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. *Proc. of the 1995 Int'l Conference on Parallel Processing*, volume I, pages 1-10, August 1995.

[58] A. Nowatzyk, G.Aybay, M.Browne, E.Kelly, M.Parkin, B.Radke, and S.Vishin. Exploiting Parallelism in Cache Coherency Protocol Engines. In *Proc. of the First Int'l Euro-Par Conference*, pages 271–286, August 1995.

[59] K. Petersen and K. Li. Multiprocessor Cache Coherence Based on Virtual Memory Support. *Journal of Parallel and Distributed Computing*, 29(2):158–178, September 1995.

[60] L.L. Petersen, N.C. Hutchinson, S.W. O'Malley, and H.C. Rao. The *x*-kernel: A Platform for Accessing Internet Resources. *IEEE Computer*, 23(5):23-33, May 1990.

[61] R.W. Pfile. Typhoon-Zero Implementation: The Vortex Module. Technical Report, Computer Sciences Department, University of Wisconsin-Madison, February 1995.

[62] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-Level Shared Memory. *Proc. of the 21st Annual Int'l Symposium on Computer Architecture*, pages 325-337, April 1994.

[63] S.K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. *Proc. of the 23rd Annual Int'l Symposium on Computer Architecture*, pages 34-43, May 1996.

[64] S.K. Reinhardt, B. Falsafi, and D.A. Wood. Kernel Support for the Wisconsin Wind Tunnel. *Proc. of the Second USENIX Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[65] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM TOMACS Special Issue on Computer Simulation*, 1997.

[66] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. *Proc. of the 23rd Annual Int'l Symposium on Computer Architecture*, 1996.

[67] A. Saulsbury, T. Wilkinson, J. Carter and A. Landin. An Argument for Simple COMA. In *Proc. of the 1st Symposium on High-Performance Computer Architecture*, pages 276-285, January 1995.

[68] D.J. Scales, K. Gharachorloo and C.A. Thekkath. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. *Proc. of the 7th Int'l Conference. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174-185, October 1996.

[69] D.J. Scales, K.Gharachorloo, and A.Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proc. of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[70] A. Seznec. A Case for Two-Way Skewed-Associative Caches. *Proc. of the 20th Annual Int'l Symposium on Computer Architecture*, pages 169-178, 1993.

[71] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus and D.A. Wood. Fine-grain Access Control for Distributed Shared Memory. *Proc. of the Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297-306, October 1994.

[72] I. Schoinas, B. Falsafi, M.D. Hill, J.R. Larus, C.E. Lukas, S.S. Mukherjee, S.K. Reinhardt, E. Schnarr, and D.A. Wood. Implementing Fine-Grain Distributed Shared Memory on Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin--Madison, February 1996.

[73] P. Stenström. A Survey of Cache Coherence Scheme for Multiprocessors. *IEEE Computer*, (23)6:12-24, June 1990.

[74] P. Stenström, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. *Proc. of the 19th Annual Int'l Symposium on Computer Architecture*, pages 80-91, May 1992.

[75] C.A. Thekkath and H.M. Levy. Hardware and Software Support for Efficient Exception Handling. *Proc. of the Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110-119, October 1994.

[76] Texas Instruments. SuperSPARC User's Guide. October 1992.

[77] J. Torrellas, D. Padua. The Illinois Aggressive COMA Multiprocessor Project (I-ACOMA). *Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computing*, October 1996.

[78] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture*, pages 392-403, June 1995.

[79] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proc. of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 279–289, October 1996.

[80] W.-D. Weber et al. The Mercury Interconnect Architecture: A Cost-effective Infrastructure for High-performance Servers. In *Proc. of the 24th Annual Int'l Symposium on Computer Architecture*. June 1997.

[81] A.W. Wilson Jr., R.P. LaRowe Jr., and M.J. Teller. Hardware Assist for Distributed Shared Memory. In *Proc. of the 13th Int'l Conference on Distributed Computing Systems (ICDCS-13)*, pages 246–255, May 1993.

[82] S. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture*, pages 24-36, June 1995.

[83] L. Yang and J. Torrellas. Speeding up the Memory Hierarchy in Flat COMA Multiprocessors. In *Proc. of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA-3)*, February 1997.

[84] L. Yang, A.-T. Nguyen, and J. Torrellas. How Processor-Memory Integration Affects the Design of DSMs. *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[85] Z. Zhang and J. Torrellas. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. *Proc. of the 3rd Symposium on High-Performance Computer Architecture*, February 1997.

[86] Y. Zhou, L. Iftode, K. Li, J.P. Singh, B.R. Toonen, I. Schoinas, M.D. Hill, and D.A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 193–205, June 1997.

[87] R.N. Zucker and J-L. Baer. Software versus Hardware Coherence: Performance versus Cost. In *Proc. of the 27th Hawaii Int'l Conference on System Sciences (HICSS-27)*, volume I, pages 163–172, January 1994.

Appendix A

COHERENCE HANDLERS BOOTSTRAP CODE

This appendix is listing the SPARC assembly code used as a prologue and epilogue for SC-COMA's coherence event handlers (the MEXC and CINT handlers).

The MEXC handler prologue/epilogue

```
        .global mexc_low
        .type   mexc_low,#function
mexc_low:
        rd      %wim, %l4
        srl     %l4, %l0, %l7     ! (wim>>cwp == 1) -> invalid window
        cmp     %l7, 1            ! sets Z if window is invalid
        bne,a   trap_setup_done2 ! Branch if not in the invalid window
        sub     %fp, (24+8)*4, %sp

! Handle window overflow
win_of2:
        save    %g0, %g0, %g0     ! Slip into next window
        std     %l0, [%sp + (0*8)]! Save L&I regs. and update WIM
        rd      %psr, %l0
        mov     1, %l1
        sll     %l1, %l0, %l0
        wr      %l0, 0, %wim
        std     %l2, [%sp + (1*8)]
        std     %l4, [%sp + (2*8)]
        std     %l6, [%sp + (3*8)]
        std     %i0, [%sp + (4*8)]
        std     %i2, [%sp + (5*8)]
        std     %i4, [%sp + (6*8)]
        std     %i6, [%sp + (7*8)]
        restore
        sub     %fp, (24+8)*4, %sp

trap_setup_done2:
        wr      %l0, 0x00000f00, %psr   ! Read the manual. First PIL=0xf,
        wr      %l0, 0x00000f20, %psr   ! ...then ET=1.
```

```
! Prepare to call the C MEXC handler
        sethi   1, %g0          ! CM2_TRAP GET_PID
        call    handle_mexc     ! handle_mexc(myid)
        sethi   %hi(pending_req),%l4

insn_emu:                       ! prepare to copy the faulting insn
        call    1f              ! puts the PC (insn_emu) into O7
        ld      [%l1], %l5      ! load the instruction
1:
        tst     %o0             ! handle_mexc() rets &spin_flag or 0
        bz      got_reply       ! skip wait_reply if return is 0
        st      %l5, [%o7+insn_slot-insn_emu]   ! store insn at insn_slot

        st      %g0, [%o0]      ! spin_flag[myid] = LOCKED
        wr      %l0, 0x00000020, %psr  ! PIL = 0 to enable msg rec.

wait_reply:                     ! wait to be unlocked by interruption
        lduba   [%o0]0,%o1      ! spin_flag[myid] -> O1
        tst     %o1
        bz      wait_reply
        nop

        wr      %l0, 0x00000f20, %psr
        nop; nop; nop;

got_reply:
! Now PIL=0xf, as set above after a reply or before calling handle_mexc

! Execute the memory access instruction.
!!!! Self-modifying code. Beware of insn prefetches!
!!!! Also, not to be shared.

        restore %g0, %g0, %g0   ! to the context of the faulting insn.
        .global insn_slot
insn_slot:
        nop                     ! overwritten by the faulting insn
        save %g0, %g0, %g0      ! back to the trap window

        ld      [%l4+%lo(pending_req)],%o1
        cmp     %o1,0           ! if (pending_req.num_buf_rq == 0)
        be,a    no_treat_buff   !    no buffered requests to handle
        wr      %l0, 0, %psr    ! Restore PSR. only if branch taken

! Handle buffered requests for the same line
        sethi   1, %g0          ! CM2_TRAP GET_PID

        call   treat_buff_req   ! %o1 has the count of buffered rq
        st      %g0, [%l4+%lo(pending_req)] ! pending_req.num_buf_rq = 0

        wr      %l0, 0, %psr    ! Restore PSR
no_treat_buff:
        mov     -1,%l0          ! pending_req.logical_ad = NOT_PENDING
        st      %l0,[%l4+%lo(pending_req+8)]
return_mex:
        RETT_NEXT
```

The CINT handler prologue/epilogue

```
        .global int_low
        .type   int_low,#function
int_low:
        rd      %wim, %l4
        srl     %l4, %l0, %l7     ! (wim>>cwp == 1) -> invalid window
        cmp     %l7, 1            ! sets Z if window is invalid
        bne,a   trap_setup_done   ! Branch if not in the invalid window
        sub     %fp, (24+8)*4, %sp

! Handle window overflow
win_of:
        save    %g0, %g0, %g0           ! Slip into next window
        std     %l0, [%sp + (0*8)]      ! Save L&I regs. and update WIM
        rd      %psr, %l0
        mov     1, %l1
        sll     %l1, %l0, %l0
        wr      %l0, 0, %wim
        std     %l2, [%sp + (1*8)]
        std     %l4, [%sp + (2*8)]
        std     %l6, [%sp + (3*8)]
        std     %i0, [%sp + (4*8)]
        std     %i2, [%sp + (5*8)]
        std     %i4, [%sp + (6*8)]
        std     %i6, [%sp + (7*8)]
        restore
        sub     %fp, (24+8)*4, %sp

trap_setup_done:
        wr      %l0, 0x00000f00, %psr   ! Read the manual. First PIL=0xf,
        wr      %l0, 0x00000f20, %psr   ! ...then ET=1.

! Prepare to call the C interrupt handler
        sethi   1, %g0                  ! CM2_TRAP GET_PID; myid->%o0

        call    handle_int              ! handle_int(myid)
         nop

        wr      %l0, 0x0, %psr          ! restore PIL to 0x0
1:
        nop                             ! Delay slot
        nop                             ! 2nd delay slot
        RETT
```