Hardware vs. Software Implementation of COMA

Adrian Moga, Alain Gefflaut, and Michel Dubois

CENG 97-03

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

January 1997

Hardware vs. Software Implementation of COMA

Adrian Moga, Alain Gefflaut*, and Michel Dubois

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
(213)740-4475
Fax: (213) 740-7290

*Siemens AG
Public Communication Networks Group
Munich, Germany

{moga,dubois}@paris.usc.edu
gefflaut@mch.scn.de

USC CENG Technical Report 97-03

Abstract

Traditionally, cache coherence in multiprocessors has been maintained in hardware. However, the cost-effectiveness of hardwired protocols 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 the addition of hardware support for fine-grain sharing.

We have developed a software protocol for a COMA (Cache-Only Memory Architecture) on a distributed network of processing elements. We call the system 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 to provide the applications and the bulk of the kernel with the same services as its hardware counterpart. The software emulation layer has been written and we compare SC-COMA to an idealized hardware COMA through detailed simulations.

Our results show that SC-COMA is competitive. On systems with 32 processors, it achieves a slowdown of 11-66% with respect to its hardware counterpart, across a 25-75% range of memory pressures. SC-COMA scales well for up to 32 nodes provided the communication-to-computation ratio is moderate. For the organization of the attraction memory, we find that four-way set-associativity is both necessary and sufficient. A study on the impact of faster processors on SC-COMA's relative performance indicates a consistent improvement, but with a limitation due to overheads stemming from the loosely-integrated, decoupled design. We conclude that SC-COMA is a viable solution, using a simple hardware addition, to transform networks of workstations into powerful multiprocessors.

Keywords: distributed shared-memory multiprocessors, COMA, software cache coherence.

1. Introduction

Several factors are motivating current research on Distributed Shared Memory (DSM) to investigate *hybrid* systems, which provide cache coherence with software-implemented protocols. In spite of achieving the best performance, hardware DSMs [17][3] are hard to build and costly, and lack flexibility. All these disadvantages are eliminated by software DSMs [18][7], at the cost of some performance degradation. 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 [2]. To fight *false sharing* [6] effects, caused by the coarse, fixed-sized units of coherence (the pages), hybrid hardware/software systems rely on fine-grain access control. Apart from the performance aspects, hybrid systems are better suited for off-the-shelf components, which further improves their cost-effectiveness.

DSM protocols are mainly influenced by the memory organization scheme, although many variations exist in the organization of the directory. NUMA protocols are simpler and the solution of choice in many systems. COMA protocols, although more complex, have the appeal of automatic data replication and migration. NUMA protocols have been evaluated previously for a variety of hybrid systems. Alewife [4] and the software-only directory proposal [8] execute coherence actions partly in a controller and partly on the main processor. Blizzard-E [26] uses exclusively the main processor, but it has only partial hardware support for fine-grain access lookup. Blizzard-S [26] and Shasta [25] use the main processor for both the protocol and the access lookup. Memory informing operations [12] provide a better mechanism to trigger protocol actions on the main processor. Dedicated processors are used as protocol engines in Typhoon [21] and FLASH [15]. The latter could support a COMA protocol but there is no evaluation available yet.

To our knowledge, this paper is the first to detail the software implementation of a COMA protocol and evaluate its performance. The contribution of the paper is to present the design of a hybrid COMA, where the protocol engine is emulated in software executed on the main processor, and to compare its performance to an idealized hardware implementation of the protocol. Our decision to study a hybrid COMA is motivated, in part, by the intuition that a COMA maximizes node hit rates, hence reducing protocol engine occupancy. Another reason is that studies involving NUMA protocols in hybrid architectures with varying degrees of hardware aggressiveness are plentiful [10][8][22].

In the current version of our system, the targeted platform is a network of uniprocessor workstations. Hence, the protocol runs on the main processor, but the solution could easily be adapted if a dedicated processor is available. Fine-grain memory access checking support and the controller for a set-associative memory are incorporated in a single, relatively simple functional unit. This could be easily integrated in the memory controller or just be plugged into the local bus, like in Typhoon-0 [22] or START-NG [5].

Our results show that a software-implemented protocol engine can perform very well, even when compared to an ideal hardware implementation. Addressing some of the concerns about performing protocol actions on the main processor [10][11], we show that efficient switching of the main processor between application and protocol can be done without hardware support for context switching. We also show that the overhead of protocol actions does not disrupt application performance to a significant degree. The protocol engine occupancy, relative to measurements in FLASH [10], is reduced by several effects. The number of accesses hitting in the local memory is maximized by the COMA organization and such accesses bypass the protocol engine, generating zero occupancy, unlike in FLASH. When the main processor is blocked in a remote access or even for synchronization, servicing external requests also produces zero occupancy.

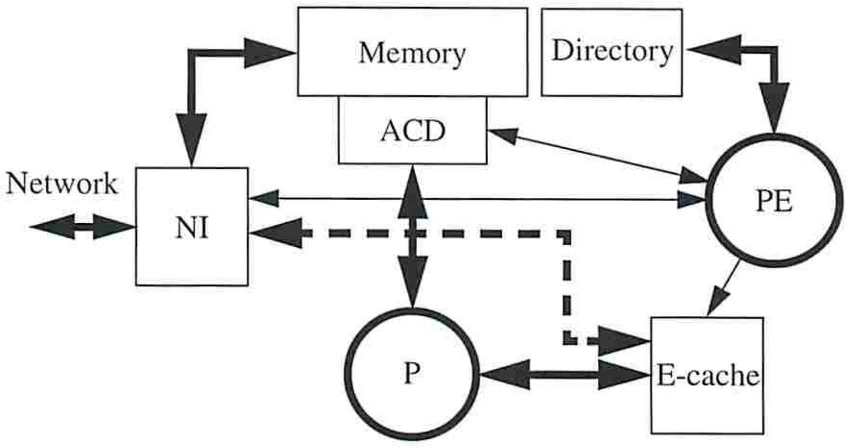
After we describe the COMA and the coherence protocol in Section 2, we explain the issues involved in supporting a protocol engine inside the main processor and we detail our solutions. The software version of COMA is compared to an idealized hardware counterpart in a setup described in Section 4. Evaluation results follow in Section 5. The impact of processor technology is debated in Section 6. We end with a discussion of related work and with our conclusions in Section 8.

2. Cache-Only Memory Architecture (COMA)

2.1. DSM hardware substrate

Figure 1 illustrates the composition of a generic DSM node. Processor P issues references, most of them satisfied by the internal and external caches. E-cache misses reach the Access Checking Device (ACD) for a possible completion of the access in the local memory. When memory contains a valid copy of the data, it sends it to the cache. Otherwise, the ACD signals a miss to the Protocol Engine (PE). The PE is responsible for completing the remote access by communicating with other nodes over the network. Request and reply packets, sent in accordance to a set of rules (the coherence protocol) and the information maintained in a distributed directory, are processed by the PEs to accomplish this task. When a reply packet contains data, the data can be transferred from the network interface (NI) buffers to the main memory or, in some cases, directly to the cache. Apart from implementation differences, the memory organization, embodied in the ACD, and the coherence protocol supported by the PE are the most important features in a DSM architecture. These two features defining the COMA will be discussed next. Note that a COMA does not require the ability to transfer data between the E-cache and the network.

Figure 1. Hardware substrate for a DSM node



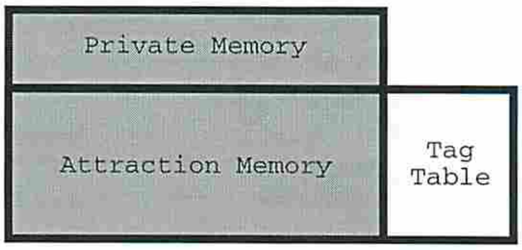
2.2. Memory organization in COMA

As shown in Figure 2, the portion of main memory hosting shared data in COMA is called the *attraction memory* (AM) and has a set-associative organization similar to a cache. A block of data in the AM is called *line* and is identified by a tag stored in the Tag Table. Along with the tag in each table entry, several bits encode the state of the line, such as *Invalid*, *Shared*, *Exclusive*. The Tag Table is queried by the ACD on every AM access from the local processor and its information is used to signal a miss to the PE. It may also be queried by the PE during the processing of external requests or AM misses. The Tag Table is updated exclusively by the PE. We assume the presence in main memory of a private space, free of tags, which hosts the stack and, in the fork model of parallel processing, the data segment and private heap as well.

The COMA memory organization comes with a trade-off. On one hand, it provides support for automatic data replication and migration in memory, a great relief for programmers and operating systems.

From a performance perspective, many cache conflict misses can be satisfied from the local memory, thus improving the node hit ratio and overall latency. On the other hand it suffers from the tag-checking overhead and the replacement problem. Searching the AM tags increases memory latency, especially for misses. When replacing a line out of memory to make room for another, it is possible that no other copies of the replaced line exist in the system. Other nodes must be asked to accept this line in their memory, complicating the coherence protocol and, indirectly, affecting system performance. To allow for data replication, the total allocated memory for the AMs should exceed application needs. The ratio between the sizes of shared data in the application and the AMs is called *memory pressure*. Memory pressure has a direct effect on the rate of replacements.

Figure 2. Logical organization of memory in a COMA



2.3. Coherence protocol

2.3.1. The directory

The directory is made of the Pointer and Copyset tables and is accessed only by the PE in the events of an AM miss or an external request. As per the Flat COMA [13] organization, the Pointer Table in the home node of a line contains the identifier of the node currently owning the line. The size of a pointer is $\log N$ bits, N being the number of nodes. The Pointer Table is indexed and dense. The owner node (not the home) for a line contains the presence bits in the Copyset Table, indicating the nodes with a copy of the line. While other presence representation methods are possible, we are assuming a full bitmap encoding. Thus, the size of a copyset is equal to the number of nodes and we need as many copysets as there are lines in the AM. The Copyset Table is indexed and sparse because not every line is owned.

2.3.2. Protocol actions

We evaluate a write-invalidate protocol, very much like the one in DASH [17], with extensions for COMA and optimizations designed to improve performance when protocol actions are executed by the main processor. Currently, it implements a sequentially consistent memory access model with no prefetching; hence, just one request can be pending in each node at a time.

Each line in the AM can be in one of four stable states: *Exclusive, Master-Shared, Shared* and *Invalid*. The *Exclusive* (writable) and *Master-Shared* (read-only) states denote ownership of the line. The owner node must ensure that at least one copy of the line exists in the system. Thus, before replacing the line, it must *inject* it into another node. On a read miss, nodes acquire a line in state *Shared*.

A *home* node is statically associated with each line, as in COMA-F [13], to hold the *owner identification pointer*. After a miss in the AM, the PE sends a request to the line's home. If home is also the current owner, it replies to the request. If not, it forwards the request to the current owner, which will reply directly, bypassing the home. As opposed to COMA-F, the line owner (instead of home) maintains the current *copyset* (presence bits) for the line. This allows the owner to send invalidations without consulting the home node and to piggyback the number of acknowledgments to be collected on the reply, thus saving a message. Acknowledgments are sent directly to the originator of the request, which is idling anyway.

Keeping the copyset with the owner also allows us to implement *request buffering*. This technique is targeted at eliminating the retry (NACK) messages used by homes when the owner pointer is locked (usually, during transactions involving a change of ownership, such as write requests). Instead, the home always updates the owner table to point to the last writer and starts forwarding new requests to it. In case forwarded requests reach the new owner before the reply from the old owner, they are buffered. After receiving the reply, the new owner responds to buffered requests. When using software to implement the PE, request buffering is very cheap to support. This strategy reduces the number of messages exchanged by nodes, as well as the number of interruptions on the main processor.

For replacements, the protocol chooses a victim line in the set according to the priorities: (1) *Invalid*, (2) *Shared*, (3) *Exclusive/Master-Shared*. The replacement of a *Shared* line is silent and does not trigger a copyset update. *Exclusive/Master-Shared* lines are sent to their home node. If the home does not accept the injection, the request is forwarded from node to node until it finally finds its place. An injected line can only replace an *Invalid* or *Shared* line and does not generate other replacements in the visited nodes. During this injection, the owner pointer for the line at the home node is locked and any request for the line is NACKed. Pending replacements are the only situations where the protocol uses NACKs.

3. A COMA with software-implemented protocol engine

We now describe a COMA having the protocol engine implemented entirely in software executed on the main processor. Effectively, the processor is multiplexed between application and protocol modes. The protocol mode is a sub-kernel mode where the processor executes a thread with system-level privileges and having access rights to special address ranges, mapped over the directory, tag table, and the NI and ACD registers. The coherence threads run without interruptions and preserve atomicity with respect to faulting references. In essence, the bare message-passing hardware and the coherence software handlers form a virtual machine having all the properties of a COMA[1]. With a careful design, this can be accomplished with very low overhead.

3.1. Mechanisms

Referring back to Figure 1, the embedding of the PE into P requires several mechanisms to be present at the processor interfaces with the ACD, NI, memory and cache to fully support the operations of the PE. The ACD access miss signal to the PE is implemented with a bus error transaction. This generates a synchronous data access exception trap, because we assume a processor with blocking loads and stores[2]. When the trap is shared by several events, such as MMU faults, status registers are queried to select the appropriate handler. The handler can retrieve information about the access, such as physical address and type, from the memory-mapped ACD registers. The processor trap is precise, so that the application can be safely restarted after the access fault.

When external requests and replies are received from the network, the NI signals the PE with asynchronous interrupts. Memory-mapped registers are used to communicate with the network interface. DMA is available for data transfers between memory and the network, to off-load the processor and to bypass the caches.

The Tag Table and the directory are stored in main memory, for simplicity. Both of them reside in

1. we are aware that further extensions toward supporting virtual memory and I/O are required.
2. store buffers and asynchronous data access traps could be accommodated by more complex fault handlers.

reserved areas. Because the ACD shares access to the Tag Table with the PE, the Tag Table cannot be cached. It could be cached if access to the Tag Table can be done in write-through mode, but we do not assume this. However, the directory can be cached and we take advantage of this.

Finally, when servicing external requests, the PE needs the ability to downgrade[3] certain lines in the cache, based on their physical address. Our assumption is that the processor and the cache controller support two special instructions: *Invalidate* and *Flush*. *Flush* is used to downgrade from *exclusive* to *shared*, possibly generating a write-back. Alternate Space Identifier (ASI) instructions in the SPARC architecture can be used to this purpose. Otherwise, the processor could program a bus device, namely the ACD, to issue the appropriate bus transactions.

3.2. The coherence software handlers

We now describe, in more detail, how the processor switches between protocol and application modes and how the coherence handlers operate. In the current design of SC-COMA, two types of coherence trap handlers are present, reflecting the client/server nature of the PE. A synchronous *memory exception* handler implements the client and starts when an access to the attraction memory cannot be served. The missing access is not completed and will be re-executed after the line is brought locally in the correct state. The asynchronous *interrupt* handler implements the server and manipulates messages containing requests and replies issued by remote processors. At the time of an interruption, the processor can be either in application mode or in a memory exception and waiting for a reply.

The bulk of the trap handlers is written in C for ease of development. A prologue and an epilogue, written in assembly language, are responsible to make the execution of the C routines transparent to the interrupted thread. When entering protocol mode, the prologue saves the current context entirely in processor registers (trap window locals) in most cases. This is made possible by SPARC's windowed registers and by forcing the compiled code for the C routines to avoid using global registers at no performance cost. The code for the prologue and the epilogue is highly optimized, adding an overhead of just 25-30 instructions to the C routines. Since this code is very likely to hit in the on-chip cache and data transfers are not involved, the transition to the C routines and back happens in less than 30 cycles on average.

3.2.1. Memory exception handler

The memory exception handler is composed of three parts. First, the processor identifies the missing line, its current state in the memory, and the type of memory access (Read, Write, LoadStoreAtomic) by reading the ACD registers. Based on this information, the processor builds a request which is sent to the home node. If the AM has to make room for the requested line, a replacement is performed after sending the request. Thus, replacements are not on the critical path. Second, the processor enters a spinning loop waiting for a reply to its request. The third part of the memory exception handler starts as soon as the processor has received the reply to the request through an interruption[4]. The processor can now restart its computation and re-execute the missing instruction. The different phases of a memory exception are depicted, in more detail, in Figure 3.

When a reply reaches a node, the pending memory access should be restarted and completed. However, between the time the interruption is completed and the time the instruction is restarted, another interruption for the same memory line could be raised and remove the line from the memory; the restarting access then triggers a new memory exception. Two or more nodes competing for the same line could hence

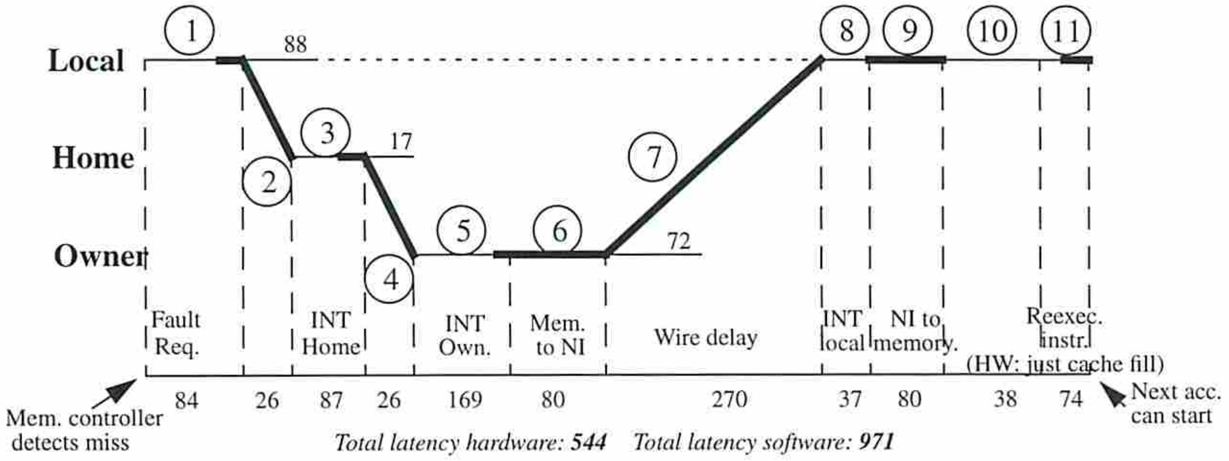
3. we assume the caches support multiprocessor protocols.
4. polling is another option.

be tied in a livelock situation where each node loses the line before the completion of its current memory access. The time window in which a memory line may be invalidated before the processor restarts accessing it has been called the *window of vulnerability* in [14]. To avoid livelocks and ensure forward progress, this window of vulnerability must be closed.

One approach for closing the window of vulnerability, proposed for hardware coherence protocols, is called *associative locking* in [6]. It consists of locking a cache line when it is loaded into the cache and deferring its invalidation until the processor effectively accesses it. When an interruption occurs and the line is locked, the request must be rejected so that the local processor can restart the access on the line before loosing it. Adapting such a solution to our implementation would require a new *locked* state for the lines of the attraction memory as well as modifications of the ACD. To minimize hardware complexity, we have implemented a software solution in SC-COMA. To close the window of vulnerability, the instruction generating the access fault is copied into the code of the memory exception handler in a dedicated slot and executed with interruptions disabled after the reply is received. The instruction executes in exactly the same context as when the fault was generated, albeit in system mode and with a different program counter. After the epilogue, execution resumes with the following instruction. A distinct advantage of our software solution over associative locking is that it cannot create any deadlocks.

Figure 3. Anatomy of a remote miss

After sending the request (1) the local processor becomes idle. The request message traverses the network to the home node (2). Upon arrival, it interrupts the home node 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 and, possibly, 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). Checking for buffered requests, restoring the application context and returning from the fault handler are the final events, but their penalty has been lumped into phase 10. Thin lines are for software latencies and thick lines are for hardware latencies in 5ns cycles..



3.2.2. The interruption handler

Interruptions are triggered by the network interface when external messages (requests or replies) are received. When the message is a request, the handler simply treats the request and sends a reply message. If the message is a reply, the handler updates the line state and, if necessary, transfers the line into memory. Finally, the handler unlocks the waiting loop in the memory exception handler. No other interruption is allowed within an interruption handler.

4. Performance Evaluation Methodology

4.1. The two architectures

The operation of the ACD is identical in terms of timings for all AM accesses, so that uniprocessor applications perform identically on the two architectures. The memory controller implements a 4-way set associative attraction memory with 128-byte lines. The Tag Table is stored in regular DRAM, like regular data. We do not assume memory interleaving or any fast page mode optimizations. The tags and states for the lines in a set are packed in a single double word (8 bytes), which is fetched and checked in a full memory cycle time, prior to the actual access. Access checking to the code and private data segments is disabled.

HW-COMA's PE is a hardware controller with zero latency. The PE has instantaneous access to the directory and the tag table. Hence, the latency of a remote miss is due exclusively to data transfer delays and contention for the local bus.

SC-COMA's PE is implemented in software. The latency of the PE operations is variable. The directory is stored in main memory, but is cacheable. Accesses to the tag table are uncached. When handlers search for a line in the AM, the ACD provides assistance, so that the cost is slightly more than an uncached read. Although we still simulate arbitration for the local bus, there is no longer contention with external requests.

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 caches (FLC) and second-level caches (SLC) are direct-mapped with 64 bytes lines. The FLC is split into two independent 16 Kbytes caches, for instructions (IC) and data (DC). The DC is write-through with no allocation on write. The SLC is four-way set-associative and unified. The SLC size must be scaled down to reflect the small data set size of the benchmarks. 64 Kbytes is enough for the primary working set WS1 of the benchmarks [30], while at the same time yielding a reasonable ratio of memory versus cache sizes. (The shared memory size per processor in these runs varies between 100 Kbytes to 2Mbytes). The SLC connects to the memory by a 128-bits local bus running at 50 Mhz. The memory has a 16-byte wide interface and an access time of 28 cycles (140 ns). The critical word of the line is fetched first. Read latencies are given in Table 1.

The network interface is controlled through a set of memory-mapped registers. A 128 byte line is transferred between network buffers and main memory in 80 cycles, assuming DMA assistance. The simulated network is a crossbar with a constant bandwidth of 100 MB/s between two nodes. Hence the transfer of a request (8 bytes) takes 16 cycles and the transfer of a message containing a line (128 bytes) takes 272 cycles. Ten cycles are added for processing at the reception.

Table 1. Read latencies

Read Requests	**Latency (pclocks)**
Read served by FLC	1
Read served by SLC	7
Read served by the local memory (64 bytes) SCC-NUMA, SS-COMA, SC-COMA private data	46
Read served by the local memory (64 bytes) SC-COMA shared data	46+28=74
Direct read access from memory(4 bytes, uncached)	40

4.2. The simulator

We have developed a flexible, modular simulation environment for hybrid and hardware DSM architectures. Common modules include a processor simulator, a simplified MMU, two levels of caches, private and shared physical memory modules, 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 application code or with the simulator modules by simply modifying some macros. The former case corresponds to hybrid DSMs. In the latter case, we simulate an ideal hardware implementation where protocol handlers take zero time to execute. The processor simulator is a SPARC interpreter. With every invocation, the simulated processor advances exactly by one memory reference. Every instruction executes in one cycle, as we do not simulate the details of the instruction pipeline. Load and store instructions are both blocking.

4.3. The benchmarks

The SPLASH-2 benchmarks are compiled on a SparcStation10 using gcc-2.7.0 -O2 and linked with the system libraries of SunOS 4.1.4. 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/resumes execution for processors, as appropriate. Other ANL macros, such as G_MALLOC, CREATE, CLOCK etc., along with all operating system stubs, employ special 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 a hybrid architecture and its hardware counterpart. Where necessary, we made small modifications to the benchmarks to comply with the FORK execution model. The characteristics of the benchmarks are given in Table 2.

Benchmark	Parameters	Shared memory used (bytes)
Barnes	8K particles	2,584,576
FFT	64K points	3,555,328
LU	512 x 512	2,113,536
Ocean	258 x 258	16,375,808
Radix	1M integers	10,051,584
Raytrace	teapot	3,407,872

Table 2: Characteristics of the benchmarks

The data placement strategy is round-robin using a page-sized (4KB) allocation unit. We should point out that an informed placement, optimized for a CC-NUMA, leads to a slightly better performance for all benchmarks. However, it is an advantage of COMA that uninformed placement strategies perform almost as well as optimized strategies. When we will compare the performance of SC-COMA to an ideal hardware implementation, a less efficient initial placement is more unfavorable to SC-COMA, as the number of interrupts, hence the software overhead, is increased due to request forwarding.

5. Simulation Results

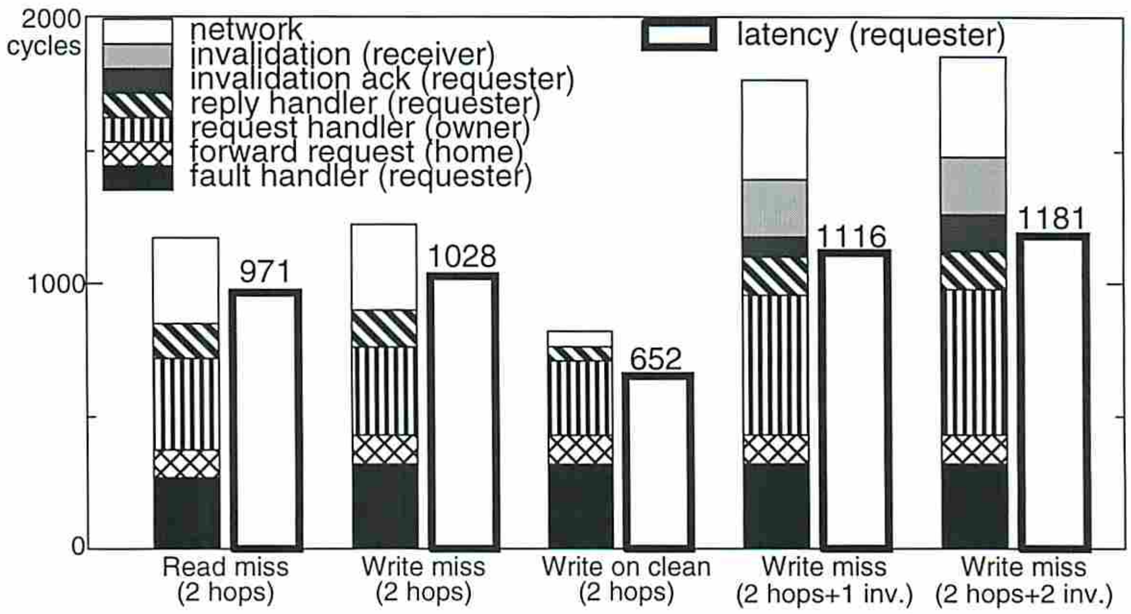
5.1. Miss Latencies

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, 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 requester consumes 65 cycles, except for the last one (76 cycles).

Figure 4. Breakdown of latencies for simple requests (micro-benchmarks) for 200 MHz processors.



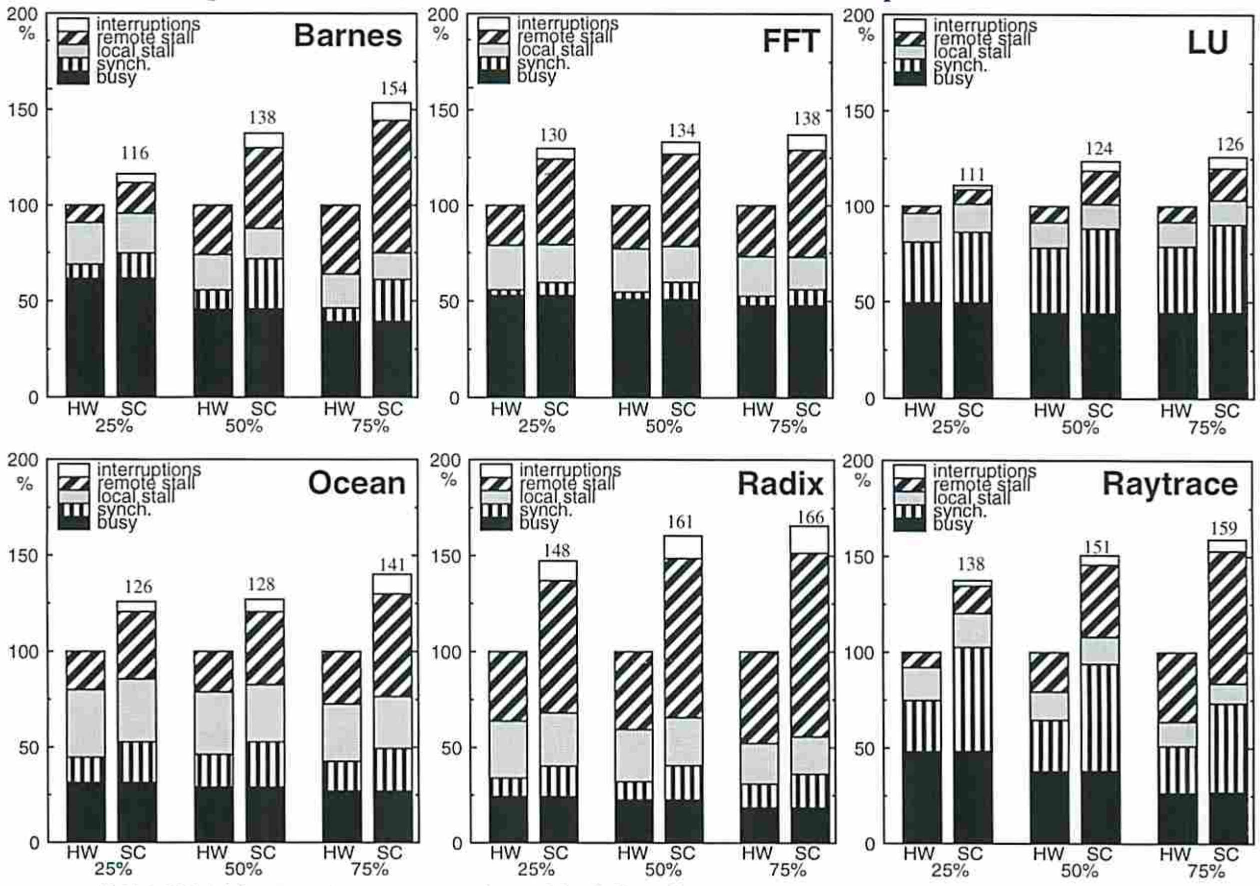
By looking at the latency times experienced by the requester (right bars for write miss cases), we see that the global cost per invalidation is between 70 and 90 cycles. This can represent a large overhead if data sharing is important. However the number of simultaneous copies of a line is usually low in typical programs and we do not think that dispatching invalidations and collecting invalidation acknowledgments by hardware, as advocated for software-extended protocols [4], would yield a large performance improvement, except for benchmarks with wide sharing. This issue is discussed further in the next section.

5.2. Execution Times

In this section we present the overall performance of SC-COMA by comparing it to an ideal hardware implementation called HW-COMA. HW-COMA uses exactly the same coherence protocol as SC-COMA, but incurs no penalty for the (software) execution of coherence actions, as if they are 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 5 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 performed 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 or a pending remote access.

Figure 5. Execution times for SC-COMA normalized with respect to HW-COMA.



SC-COMA's slowdown ranges from 11-48% at low memory pressure to 26-66% at high pressure. The local stall, in spite of almost identical cache and memory hit ratios for the two architectures, is higher in HW-COMA. This is explained by the increased delay for cache misses, when HW-COMA's controller is busy with external requests. By contrast, in SC-COMA, the cache controller has exclusive access to the bus and the memory controller.

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 3. 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 along with memory pressure.

Indirectly, the activity of the coherence handlers negatively affects the synchronization stall. The increased synchronization penalty in the context of software-implemented protocols has been attributed by Grahn and Stenström [8] 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 underutilized. 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, again overlapping some of the overhead with a blocked time.

5.3. Speedups

In Figure 10 we present speedups for up to 32 processors. The algorithmic speedup is derived for a perfect memory system, with zero stall. To gain insight into the behavior of the COMAs, we show the speedup at three different memory pressures. The total amount of memory in the system is kept constant and is divided equally among the processors. In all configurations, processors have a 64KB cache. 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.

Let's introduce a simple model to discuss the speedup. Assume the execution times using P and 2P processors are given by:

$$t_P = \frac{B}{P} + L\frac{T}{P} \qquad t_{2P} = \frac{B}{2P} + L\frac{T(1+\alpha)}{2P} \tag{1}$$

B is the total amount of computation, *T* is the inherent total traffic, *L* is the communication latency and α is a factor describing the increase in traffic from *P* to *2P* processors. The speedup t_P/t_{2P} is then given by:

$$S_{2P} = \frac{2}{1 + \frac{\alpha}{1 + \frac{B/T}{L}}} \tag{2}$$

We can infer that the speedup is ideal when the traffic stays constant ($\alpha=0$). The latency L is the only architecture-dependent factor. The smaller the L, the bigger the speedup. B/T is the computation-to-communication ratio. A large B/T would make variations of L less significant. This model does not include the effects of synchronization. In general, larger latencies exacerbate the penalties of synchronization. For a COMA, the above effects are visible at low memory pressure. With increasing memory pressure, replacements and capacity traffic start to interfere. In HW-COMA, replacements act indirectly, by increasing contention and the average latency L. However, they should have a relatively mild impact. In SC-COMA, replacements have a direct effect, as well, by interrupting application processing. Capacity traffic slows down both implementations, but especially SC-COMA.

We discuss the speedups by analyzing the communication-to-computation characteristics, as indicated by the traffic (bytes per instruction [30]), the influence of replacements and synchronization limitations. The presence of remote latency makes HW-COMA's speedup diverge from the algorithmic speedup and SC-COMA's speedup diverges even more. When communication scales well, the total traffic is roughly independent of the number of processors and the speedups should not saturate. Otherwise, stalls due to communication can become dominant and the speedup saturates. In this case, due to the higher remote latencies, SC-COMA would saturate even faster. Indirectly, increasing communication may affect synchronization penalties, making saturation even more prominent. A higher memory pressure can increase the capacity traffic. When data is mostly-read, this doesn't dramatically affect the rate of replacements.

Barnes has a small comm-to-comp ratio, resulting in a nice speedup at 25% memory pressure. At higher pressures, the capacity traffic, albeit for mostly-read data, increases sharply. In a snowball effect, synchronization penalties increase as well.

FFT has a very good algorithmic speedup and practically no false sharing. The traffic is moderate and remains almost constant for any number of processors. Furthermore, the capacity traffic and the replacements are not very much affected by the memory pressure, because there are few processor cache capacity misses.

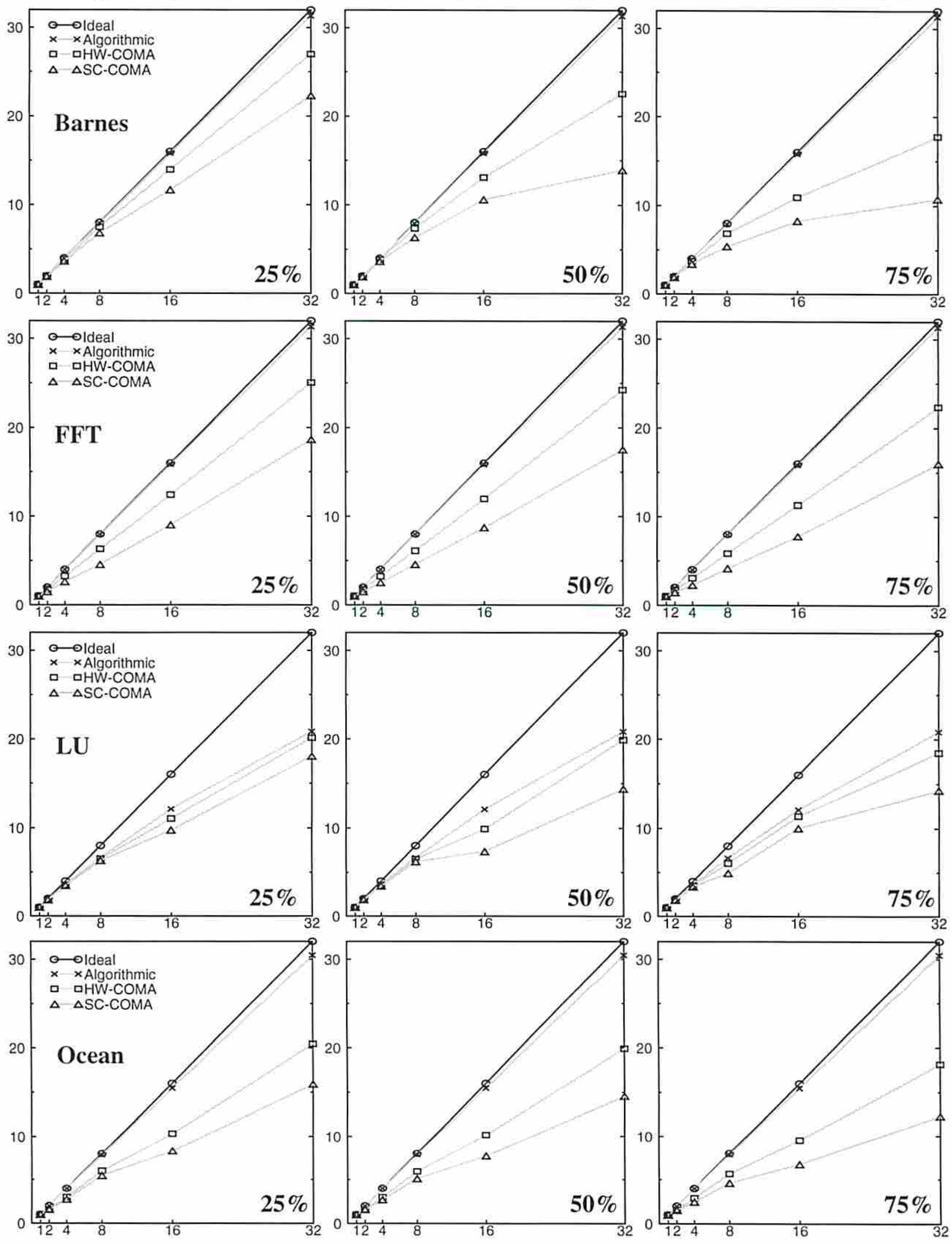
LU has the smallest comm-to comp ratio, resulting in speedups for the COMAs which are very close to the algorithmic speedup. Unfortunately, the barrier synchronization penalty for the algorithm is high and becomes even higher for the COMAs when memory pressure is raised, although the capacity traffic changes increases slowly.

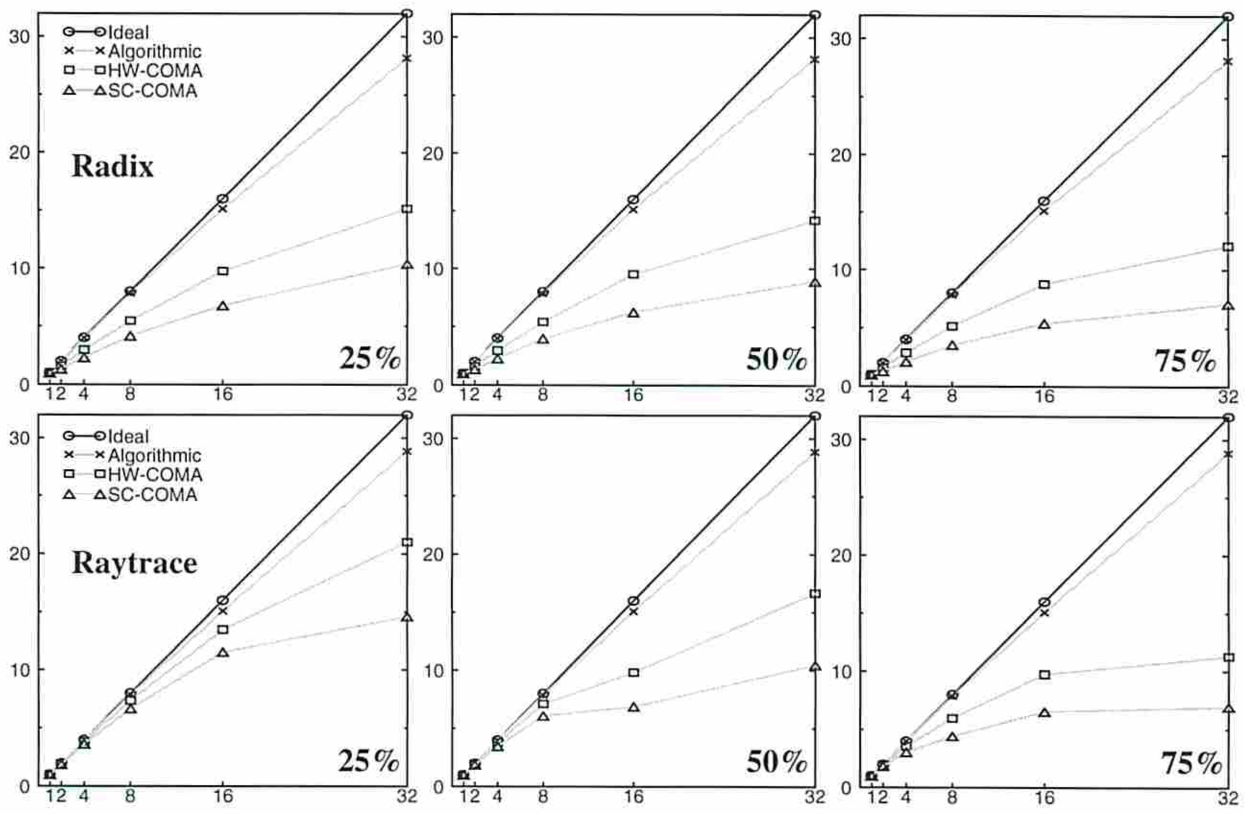
Ocean has a fairly high comm-to-comp ratio. Both capacity traffic and replacements increase moderately at 50% memory pressure, but start to pick up at 75%.

Radix has the highest comm-to-comp ratio. The capacity traffic is also significant at high memory pressure and, because most of the data is writable, there are many replacements as well. All together, these effects contribute to the worst speedup.

Raytrace has a moderately small comm-to-comp ratio. The speedup would be very good at low memory pressures if synchronization penalties were not in the way. With higher pressures, the capacity traffic increases significantly. However, replacements stay low, because data is mostly-read.

Figure 6. Speedups for up to 32 processors (three memory pressure points for each benchmark)





5.4. Effects of the associativity degree in the attraction memory

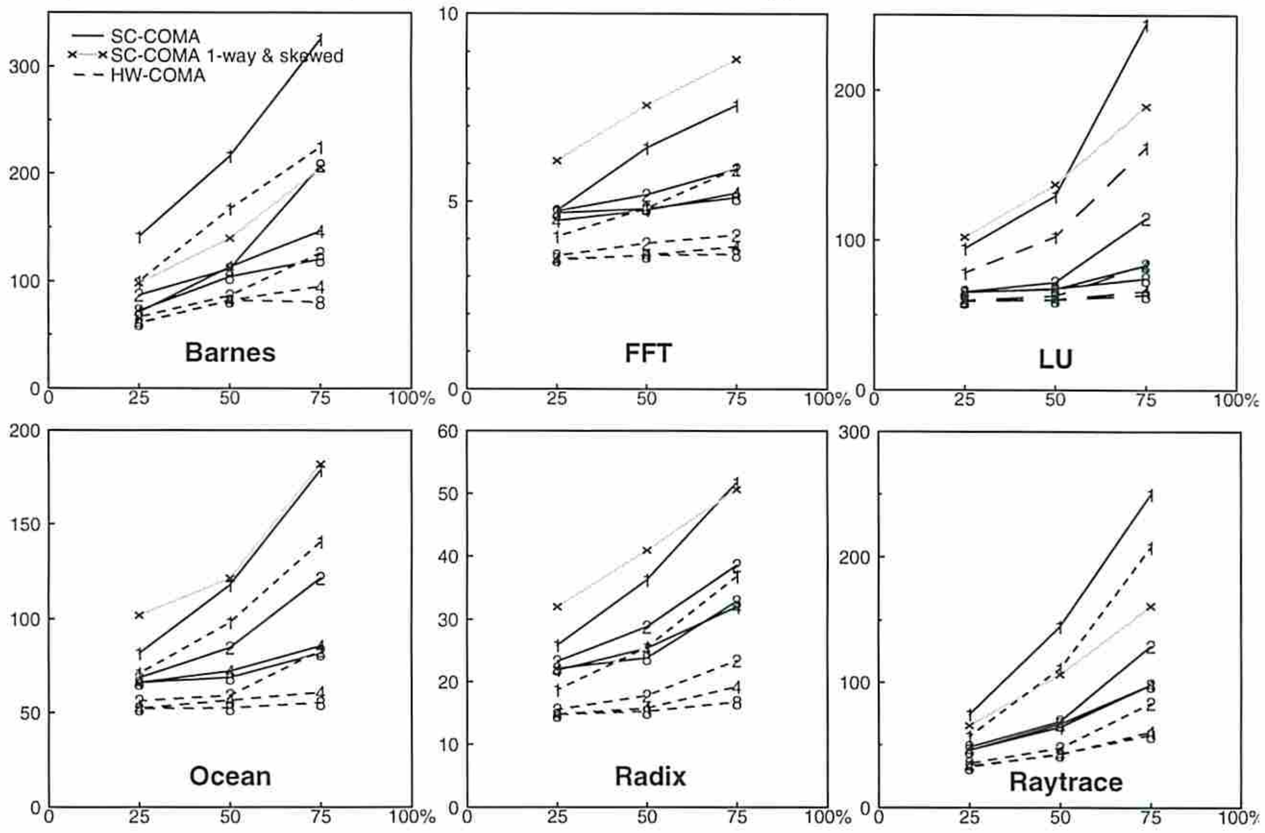
The associativity degree of the attraction memory has a double impact on the overall performance. Firstly, the AM hit ratio generally improves as the associativity is increased, resulting in fewer accesses requiring software intervention. As a side effect, the number of replacements is also reduced when associativity is increased, again reducing software overhead and network traffic. Secondly, for a direct-mapped AM, it is possible to eliminate the tag-checking overhead, which speeds up all memory accesses. Figure 7 illustrates the impact of the associativity on the execution time at different memory pressure points. The choice of a four-way set-associative AM 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 pressure increases, especially for Barnes and Ocean, the performance degrades considerably.

The bad performance of the direct-mapped AM, 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 L1 cannot be efficiently shared by many nodes, especially at a high pressure, because line L2, replaced when L1 is replicated, will likely conflict again with L1 in the node where the replacement request is sent. Skewing [27][19] 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 N1, 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. 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 7, we present the performance of a skewing scheme with four different hashing functions. The hashing functions are described by:

$$\text{set} = ((A_{19}..A_0) \text{ xor } (A_{19}..A_{15} << (N\%4))) \ \% \ \text{NUM\_SETS} \qquad (3)$$

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, as yet, whether other skewing schemes would bring significant changes, as we are still investigating this aspect. At this point, it seems that, with or without skewing, a direct-mapped AM is not an attractive option.

Figure 7. Execution times (10^6 cycles) as functions of the memory pressure for different AM associativities.



5.5. Effects of 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, from 100MHz to 1000MHz. These simulations are performed at 75% memory pressure, where the software overhead is higher, due to more frequent replacements, and the impact of faster processors is more significant. Our indicator is SC-COMA's slowdown, the ratio of execution times t_{ex}^{SC}/t_{ex}^{HW}. Indeed, as shown in Figure 8, there is an obvious trend for a relative improvement of 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 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 of the overly optimistic timings of HW-COMA, which is assumed to have instantaneous access to all the coherence tables and the directory, whereas SC-COMA must perform costly uncached accesses. Table 3 lists the estimated value where SC-COMA's slowdown is converging.

Figure 8. SC-COMA's slowdown at 75% memory pressure for different processor speeds

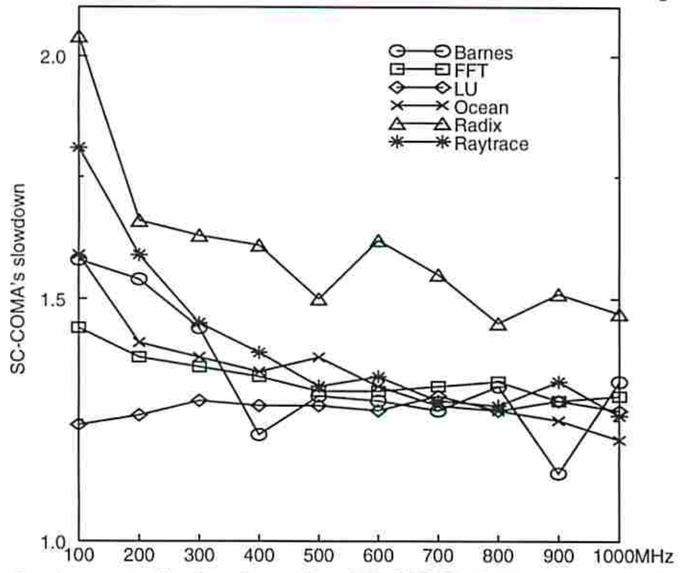


Table 3: Asymptotic slowdown for SC-COMA (estimated at 1GHz)

Barnes	FFT	LU	Ocean	Radix	Raytrace
1.32	1.30	1.27	1.21	1.46	1.26

Ar first, it was very surprising to us that the slowdowns in Table 3 were so close asymptotically, given the mix of applications. To explain why, we use a simplified model, in which the execution time of an application is given by

$$T_{ex} = T_{comp} + N_{cache}t_{cycle} + N_{mem}t_{mem} + N_{remote}t_{remote} \quad (4)$$

T_{comp} is the execution time of instructions with no memory access. N_{cache}, N_{mem}, and N_{remote} are the numbers of memory accesses that hit in the cache, hit in the local memory after a cache miss, and miss in the local memory, respectively. Given our assumptions, the latency for accessing local (t_{mem}) is unaffected by the processor's cycle time (t_{cycle}). This holds true for the average latency of a remote memory access (t_{remote}) in a hardware implementation as well. With software coherence, t_{remote} contains a component dependent on t_{cycle}, but it virtually disappears at high processor speeds. As processor speed increases, the T_{comp} and $N_{cache}t_{cycle}$ components of the execution time become less and less significant and the following approximation can be used:

$$T_{ex} \rightarrow N_{mem}t_{mem} + N_{remote}t_{remote} \quad (5)$$

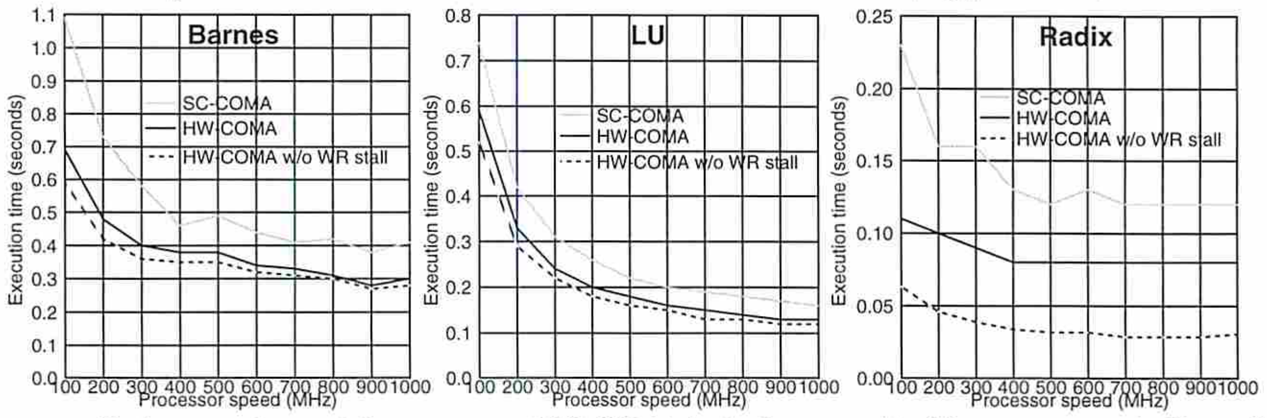
For high memory pressure, in most cases, the memory hit ratio is 35-65%, thus making N_{mem} and N_{remote} have the same order of magnitude. Because $t_{remote} >> t_{mem}$, we can further approximate:

$$T_{ex} \rightarrow N_{remote}t_{remote} \quad (6)$$

For a given application, the ratio of the execution times from equation (4), T_{ex}^{SC}/T_{ex}^{HW}, indicates the precise value of SC-COMA's slowdown, whereas the approximation from equation (6) explains why, in most applications, the slowdown converges to the value $t^{SC}_{remote}/t^{HW}_{remote}$. This is the ratio of the *average* remote latency for SC-COMA, when the overhead of executing software handlers is negligible, and that of an ideal COMA. Remote accesses due to read and write misses have very similar latency, dominated by the delays of transferring data over the network, bus and memory. Remote accesses due to ownership misses have a much smaller latency. When the mix of remote accesses does not have a very significant component of ownership misses, a frequent case, the asymptotic slowdown can be simply approximated by the ratio of the unloaded read/write latencies. This explains why, in most cases, the slowdown converges roughly to an application-independent value. This value is not equal to one, because of the overheads of programming the network interface and updating the tag and state table with uncached operations in SC-COMA. The penalty of uncached accesses is constant, regardless of processor speed.

Further comments are due to explain the behavior of Ocean, LU and Radix. SC-COMA's slowdown for Ocean converges to a lesser value than all other applications. This is because the attraction memory hit ratio is still very high (84%) even with 75% memory pressure. This makes the $N_{mem}t_{mem}$ component of T_{ex} large enough to bring the asymptotic slowdown closer to one. In LU, the slowdown seems to become larger with increasing processor speeds. Within our model, it can be proven that this phenomenon appears every time the overall node miss ratio of an application falls below a certain threshold. For the six applications we have examined, only LU fulfills this condition. By contrast, Radix has the highest node miss ratio. This makes the convergence of the slowdown become slower.

Figure 9. Execution times for SC-COMA and HW-COMA with varying processor speeds



To better understand the prospects of SC-COMA in the future world of fast processors, in Figure 9 we plot the absolute execution times for some benchmarks using processors clocked up to 1GHz. The results for the other benchmarks are similar to Barnes and LU. 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 hardware 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 a software COMA, at virtually no costs. At higher processor speeds, this advantage disappears, as the execution time starts to saturate due to memory and network latencies. The good news is that, at saturation, the performance of SC-COMA is relatively close (within 30%) to HW-COMA's, regardless of application specifics. This confirms the expectation that an ultimate performance limitation for hybrid DSMs is the efficiency of data movement and the overhead to control this movement.

The current version of SC-COMA runs on a sequentially consistent hardware. Thus, the store buffers are disabled. In Figure 9, 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. This would involve the software ability to recover pending stores from the buffer (address and data), after they are faulted, and to complete them, using untranslated stores.

It is obvious that future processors will incorporate features to fight the memory wall [24], such as bigger on-chip caches, simultaneous multithreading, 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 would these processor features affect the performance of hybrid DSMs. Would they 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 [24] and memory feedback mechanisms [12] impose at least 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, could start the appropriate handlers on the main processor.

6. Possible improvements

The current protocol for SC-COMA is just a basic version, still to be improved. The average remote access latency for SC-COMA can be reduced by using *ownership hints* [1]. Whenever the owner and home nodes for a line do not coincide, transactions require three hops (and a costly interrupt at the home node for SC-COMA). When the owner can be guessed correctly, the home is bypassed and two hops are sufficient. Previous research for hardware-implemented protocols indicated the costs of incorporating hints in the protocol offset their efficiency [1]. This is likely to change in a software implementation.

Another avenue for improving the performance of SC-COMA is provided by application-specific protocols. Some communication patterns in certain applications are better suited for write-update protocols or bulk data transfer. This can be either indicated by the programmer or detected by adaptive protocols. Adaptive sequential prefetching can also be incorporated.

7. Related Work

Research on the hardware implementations of COMA begun with the DDM [9], which used hierarchical directories. As demonstrated by the KSR-1 [3], hierarchical directories increase the latency of remote accesses. This would become even worse with software-implemented directory management. The Flat COMA, COMA-F [13], was proposed to eliminated hierarchical directories. More research on the properties of COMA was done by the DICE group [16]. To our knowledge, there is no working prototype of a Flat COMA. The Illinois Aggressive COMA (I-ACOMA) [29] group is currently building one. A variation of the COMA, using a page grain in the allocation policy, is the Simple COMA [23]. S3.mp [20] has provided a testbed for its implementation.

A variety of DSM systems have used the main processor for protocol processing. The Alewife project [4] was the first to experiment with software extensions to a NUMA protocol implemented mostly in hardware. The software protocol actions were written in C and executed on the main processor, which was providing support for very fast context switching.

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 a NCC-NUMA substrate [8]. 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. The necessity to switch context while an access is pending requires the provision of high-availability interrupts.

Another project using the main processor to execute protocol actions is Blizzard-E [26]. A cross between VSM and hybrid systems, it uses the CM5 virtual memory system to trap shared writes and the memory error-correcting codes (ECC) to implement a memory line valid bit and support fine grain sharing for read-only pages. Protocol actions run at user level, taking advantage of CM5's user-level network interface. The overhead of traps through the kernel has been reduced by careful recoding. Faulted accesses are re-executed after the trap handler terminates. There is no detail on how forward progress is guaranteed. Depending on the application, Blizzard-E is slower than KSR-1 by a factor up to seven [26].

All Distributed Virtual Shared Memory systems implement the coherence protocol in software running at user-level on the main processor, albeit using coarse-grain sharing. More recently, user-level software-only protocols with support for fine-grain sharing have been proposed by Blizzard-S [26] and Shasta [25]. They do, however, incur a penalty for performing access checking in software for every potential access to shared memory. This category of user-level software-only DSM has COMA-like features, by accumulating the working set in the local memory. However, the replacement algorithm must dispense of a whole page, which could be too large a grain in some cases, and adds the overhead of page faults.

Typhoon [21] is one of the first proposals for hybrid DSMs with high levels of integration. Its network device contains a processor dedicated to user-level protocol handling. Coherence events, snooped from the bus or signaled by the network interface, invoke user-level procedures. Bus addresses must be passed through a reverse TLB which adds to the complexity. The memory organization is inspired from Simple COMA.

The Stanford FLASH [16] is the most aggressive proposal for a DSM with software-implemented coherence. A custom node controller, the MAGIC chip, provides pipelined data paths between the processor, memory, network and I/O. It also contains the processor running the coherence protocol along with its own caches. Currently, like SC-COMA, FLASH does not allow user-level handlers and runs protocol handlers at system-level. Although a COMA protocol is planned for evaluation, the only reports [10] available are for a NUMA protocol with a directory structure using dynamic pointer allocation.

Typhoon-0 [22], START-NG [5] run the protocol on one of the processors in a standard SMP cluster. Fine-grain sharing is supported by the addition of a custom access checking device on the local bus. Both run user-level protocols. While Typhoon-0 uses a Simple COMA memory organization, START-NG manages a level-3 cache in software, requiring intervention on all level-2 cache misses.

8. Conclusions

We have presented a COMA architecture with the coherence protocol executed in software on the main processor. The hardware substrate is very close to a generic network of workstations. We rely on a custom hardware device, acting at the local bus level of every node, to organize and control a fine-grained attraction memory using standard DRAMs. Misses in the attraction memory are faulted on the bus and trigger protocol actions on the main processor. Packets received from the network are flagged with asynchronous interrupts and are handled similarly. The protocol handlers run in kernel mode and are lightweight. These handlers can be integrated into a standard OS kernel with minor modifications. SC-COMA's approach and other optimizations allow it to achieve a software overhead of just 430 cycles for a three-hop remote read.

The performance of SC-COMA compares favorably with an idealized hardware-implemented COMA. Execution times on 32 nodes for six benchmarks indicate a slowdown of 11-48% at 25% memory pressure and 26-66% at 75% memory pressure. SC-COMA scales well up to 32 processors. We studied the effects of the associativity factor for the attraction memory and concluded that, for both implementations, at least four is necessary. An associativity of four was also found to be sufficient for all benchmarks. An attempt to simplify the attraction memory controller by using a direct-mapped organization with skewing produced mixed results. Our investigation on the effects of faster processors, relative to memory and network speeds, revealed that SC-COMA's slowdown is reduced as the overall contribution of the software overhead to the remote latency shrinks. However, the slowdown cannot pass below a certain threshold due to SC-COMA's loose integration of the protocol engine with the network interface and the Access Checking Device. The results we have presented are encouraging given the simplicity of the current protocol. We are expecting improvements from further optimizations and extensions.

9. References

[1] M. Björklund, F. Dahlgren, P. Stenström. Using Hints to Reduce the Read Miss Penalty for Flat COMA Protocols. *Proc of the 28th Hawaii International Conference on System Sciences*, pp. 242-251, 1995.

[2] W.J. Bolosky. Software Coherence in Multiprocessor Memory Systems. PhD. Thesis. University of Rochester, 1993.

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

[4] D. Chaiken and A. Agarwal. Software Extended Coherent Shared Memory: Performance and Cost. *Proc. of the 21st Int. Symposium on Computer Architecture*, pp. 314-324, May 1993.

[5] Derek Chiou et al. StarT-NG: Delivering Seamless Parallel Computing. *Euro-Par'95*, Aug. 1995.

[6] M. Dubois, J. Skeppstedt, and P. Stenström. Essential Misses and Data Traffic in Coherence Protocols. *Journal of Parallel and Distributed Computing*, Vol. 29, No. 2, pp. 108-125, September 1995.

[7] 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. Symp. on Computer Architecture*, pp. 144-155, 1993.

[8] H. Grahn and P. Stenström. Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors. *Proc. of the 22nd Annual International Symposium on Computer Architecture.* Santa Margherita, Italy, June 1995.

[9] E. Hagersten, A. Landin, and S. Haridi. DDM--A Cache-Only Memory Architecture. *IEEE Computer*, Vol. 25, No. 9, pp. 44-54, Sept. 1992.

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

[11] 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.

[12] M. Horowitz, M. Martonosi, T.C. Mowry and M.D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. *Proceedings of the 23rd Annual Symposium on Computer Architecture*, pages 260-270, 1996.

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

[14] J. Kubiatowicz, D. Chaiken and A. Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM Sigplan Notices, Volume 27, Number 9, September 1992.

[15] J. Kuskin and D. Ofelt and al. The Stanford FLASH Multiprocessor. *Proc. of the 21st Annual International Symposium on Computer Architecture.* April 1994.

[16] G. Lee et al. DICE Prototype Specification. DICE TR no.6, Dept of EE, Univ. of Minnesota, Aug. 1993

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

[18] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. *Proc. of the Int. Parallel Processing Conference* pp. 94-101, 1988.

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

[20] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. *Proc. of the Int. Parallel Processing Conference*, pp. I-1-I-10, 1995.

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

[22] S.K. Reinhardt, R.W. Pfile, D.A. Wood. Decoupled Hardware Support for Distributed Shared Memory. *Proc. of the 23rd International Symposium on Computer Architecture*, May 1996.

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

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

[25] D.J. Scales, K. Gharachorloo and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, 1997.

[26] 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 6th International Conference on Architectural Support for Programming Languages and Operating Systems.* October 1994.

[27] A. Seznec. A Case for Two-Way Skewed-Associative Caches. *Proc. of the 20th Int. Symp. on Computer Architecture*, pp. 169-178, 1993.

[28] P. Stenström, T. Joe and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA architectures. *Proc. of the 19th Annual Symposium on Computer Architecture*, pages 80-91, May 1992.

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

[30] S. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. of the 23rd Int. Symp. on Computer Architecture*, pp. 24-36, 1995.