Design and Evaluation of a Software-Controlled COMA

Alain Gefflaut*, Adrian Moga, Jaeheon Jeong, and Michel Dubois

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

*INRIA/IRISA, Campus de Beaulieu
35042 Rennes Cedex
France

{gefflaut, moga, jaeheonj, dubois}@paris.usc.edu

Abstract

Traditionally, cache coherence in multiprocessors has been maintained in hardware with the support of a snooping protocol. However, the cost-effectiveness of building machines with hardwired protocols has recently been questioned. Research in Virtual Shared Memory systems, in which the software takes advantage of the virtual memory system of the kernel to implement sharing, has shown that, even with the large latencies of kernel calls, software-based shared-memory is a viable alternative.

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 coherence controllers are emulated by software executed on the node processor. Contrary to VSM systems, SC-COMA does not rely on kernel services and runs directly on the hardware, an approach we call Direct Software Emulation (or DSE) of shared memory. The software emulation layer has been written and we compare SC-COMA to CC-NUMA on the same hardware through detailed simulations of SPLASH-2 benchmarks.

Overall, the latencies experienced by SC-COMA are intermediate compared to the ones observed in VSM systems and in CC-NUMAs, and SC-COMA shows very good performance both in absolute terms and relative to CC-NUMA. Evaluating possible hardware and software optimizations, we find that no single improvement could dramatically increase the performance of SC-COMA by itself.

Keywords: shared-memory multiprocessor systems, execution-driven simulation, software cache coherence.

DESIGN AND EVALUATION OF A SOFTWARE-CONTROLLED COMA

Abstract

Traditionally, cache coherence in multiprocessors has been maintained in hardware with the support of a snooping protocol. However, the cost-effectiveness of building machines with hardwired protocols has recently been questioned. Research in Virtual Shared Memory systems, in which the software takes advantage of the virtual memory system of the kernel to implement sharing, has shown that, even with the large latencies of kernel calls, software-based shared-memory is a viable alternative.

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 coherence controllers are emulated by software executed on the node processor. Contrary to VSM systems, SC-COMA does not rely on kernel services and runs directly on the hardware, an approach we call Direct Software Emulation (or DSE) of shared memory. The software emulation layer has been written and we compare SC-COMA to CC-NUMA on the same hardware through detailed simulations of SPLASH-2 benchmarks.

Overall, the latencies experienced by SC-COMA are intermediate compared to the ones observed in VSM systems and in CC-NUMAs, and SC-COMA shows very good performance both in absolute terms and relative to CC-NUMA. Evaluating possible hardware and software optimizations, we find that no single improvement could dramatically increase the performance of SC-COMA by itself.

1. INTRODUCTION

Modern, high-performance processors rely on caches for efficient memory accesses. When connected together in a shared-memory configuration, processors with caches must interact with each other to maintain the integrity of shared writable data. Traditionally, coherence has been maintained by a hardware protocol. Typically, a small number of processors are connected to a high-speed, high-bandwidth bus and caches remain coherent by watching broadcast transactions on the bus. Directory-based protocols, which rely on point-to-point interconnects, are an alternative to these snooping protocols for connecting very high-speed processors in either small or large configurations.

In such architectures the hardware protocol provides a fine-grain access control to mem-

ory, limiting the penalties of misses and coherence transactions, as well as the impact of false sharing between the nodes. However, hardware implementations are complex and cannot be changed or improved easily. The result is a long development cycle, a higher cost of the architecture, and poor flexibility of the implementation. Since processor speeds are practically doubling ever other year, ad-hoc hardware approaches, which lengthen machine development cycle, have recently been under intense scrutiny. Even if a hardware protocol increases the multiprocessor speed by a factor of two, its overall benefit is in doubt if it delays the introduction of a multiprocessor by a year. Hardware techniques such as relaxed consistency models which require extensive software modifications are even more questionable in this context.

Because large-scale hardware-coherent shared-memory multiprocessors are so hard to build, large-scale multiprocessors have adopted a distributed memory architecture, mostly relying on message-passing for interprocessor communication. The simplest approach currently advocated is to connect independent commodity workstations with a commodity ATM switch and to port a Virtual Shared-Memory System (VSM) on top. Virtual Shared Memory (VSM) systems such as Ivy [17], Tread Marks [9] or Blizzard [23] implement shared memory with commodity hardware associated with software-only protocols. These systems replicate shared pages in the memory of each processing node and exploit the address translation hardware to control the access to shared pages and detect page misses. Traditionally, the performance of VSM systems has been quite poor, in particular because of their long latencies and of their large granularity of data sharing. The long latencies are the consequence of the software approach, which is typically based on a user-level implementation on the host operating system thus requiring costly system calls to send and receive messages and to access and modify page tables entries. The large granularity of sharing at the page level affects not only the miss latencies, but, more importantly, the amount of communication between the nodes caused by false sharing [7].

In contrast with their hardware counterpart, software-based systems are highly flexible and some of them fight the long remote memory latencies with more complex coherence proto-

cols such as adaptive protocols or very weak memory consistency models. To combat the false sharing problem, multiple-writer protocols allowing multiple nodes to simultaneously modify the same page [9] are preferred because they alleviate the mismatch problem between the page size and granularity of sharing of applications. Weak consistency models and multiple-writer protocols are presently the favored approach for networks of workstation. These implementations end up being much more complex than simple coherence protocol implementations and rely on careful and tedious application programming.

The purpose of this paper is to investigate a solution in between these two extremes. With the minimum hardware support possible, we want to implement an efficient shared-memory on top of a set of separate processing nodes communicating through message passing. As shared-memory efficiency is highly dependent on the granularity of data sharing, we base our design on a low grain detection of sharing implemented in hardware by the memory controllers of the nodes. However the work of the memory controller is limited to this detection task and the whole coherence protocol is implemented in software. The result is an architecture where most of the hardware requirements are already provided by current computers. The coherence protocol is completely implemented by software and keeps the flexibility advantages of a software approach without incurring its large performance overhead.

In the following, we give the rationale for our approach and briefly describe the hardware substrate that will support our shared-memory abstraction. In Sections 4 and 5, the software handlers and the protocol are specified. Simulations results are then reported and discussed in Section 6. Finally, in Sections 7 and 8 we compare our approach with others and conclude.

2. MOTIVATIONS AND RATIONALE

In this section we explain the rationale for our choices. First of all we believe that software protocols can be efficient, versatile and cost-effective in an environment where technology is evolving extremely fast. Second, we prefer to target a message-passing, distributed platform for hardware simplicity. Third, we do not rely on a kernel to support the software protocol; rather the software

protocol is run directly on top of the hardware substrate. Finally, we have chosen to emulate a COMA (or Cache-Only Memory Architecture) [11] and we call the system SC-COMA (or Software-Controlled COMA) to emphasize that the functionality of the protocol controllers is emulated in software.

2.1. Software versus Hardware Protocols

Even if a software implementation has larger remote miss latencies, several reasons make us believe that software coherence protocols can be very competitive with hardware coherence protocols. First, with the high network latencies of distributed systems, the software overhead time will not necessarily be the main factor in the overall communication time, hence hardware acceleration may not be really useful in this context. Moreover, fast context switching processors can be used to minimize this time and, as processor improvements continue to overcome network improvements, the time spent in software handlers will become relatively smaller. Second, software protocols give the opportunity to implement different types of optimizations (weak consistency models, adaptive protocols) that cannot, or are very hard to reliably implement in hardware. Third, multiprocessor architectures with very powerful processor nodes will be used to run applications with very large data set sizes, where the computation-to-communication ratio should be high. Finally, for the few applications which display high degrees of sharing, software approaches involving algorithm designers, programmers and compilers can limit the amount of communications in programs, thereby reducing the overhead of coherence protocols.

2.2. Message-passing versus Shared-Memory Hardware Substrate

We have chosen a message-passing substrate for our SC-COMA, because we feel that message-passing environments are much more prevalent than shared-memory environments and are easier to build. Current large-scale multiprocessors, such as Ncube/2, Intel Paragon and IBM SP-2 series [12] are message-passing. Some VSM projects start with a NCC-NUMA (non-cache-coherent non-uniform memory access) substrate, which means that the memories on all the boards are

accessible globally. An NCC-NUMA hardware substrate such as the Cray T3D [12] must be specifically designed and these machines have become scarce. An NCC-NUMA substrate can also be built from separate processing nodes in the context of network of workstations or PCs, provided complex interfaces capable of reading from and/or writing into remote memories without interrupting the remote processor [2]. Whether such interface will be cost-effective is still an open question.

2.3. Direct Software Emulation (DSE) of Shared-Memory

Traditionally, VSM systems have relied on the operating system to provide basic services such as memory allocation. In our approach, which we call *Direct Software Emulation*, the software emulation layer is between the hardware and the application or system software. Low-level memory exception traps are triggered by simple hardware mechanisms to emulate the shared-memory protocol. Shared-memory DSE was explored in the Alewife project [4] as well as by Grahn and Stenstrom in [10] in the context of a NCC-NUMA substrate. Figure 1 illustrates the difference between direct software emulation in SC-COMA and Alewife on one hand and a typical VSM system on the other.

In contrast with direct software emulation, messages sent by the emulation layer, and traps and interrupts triggered by the hardware must traverse the kernel layer in the case of virtual shared memory systems. This crucial difference has three implications. First, the latencies of memory transactions are much longer in the VSM case. These large latencies of VSM systems can be cut by carefully recoding parts of the kernel as was done in Blizzard [23]. However, they remain way above the latencies of Direct Software Emulation, as we will see in Sections 6 and 7. In this respect, the fact that VSM systems such as Blizzard-E and Tread Marks [9] exhibit competitive performance levels as compared to hardware coherence is very encouraging. Second, VSM systems let users define their own protocols whereas Direct Software Emulation defines a system-level protocol shared by all codes running on the machine, including the kernel. Third, the software emulation code in the VSM can take advantage of O/S services, such as virtual memory

management, contrary to the Direct Software Emulation, which must avoid using any hardware needed by the kernel such as the T.L.B.

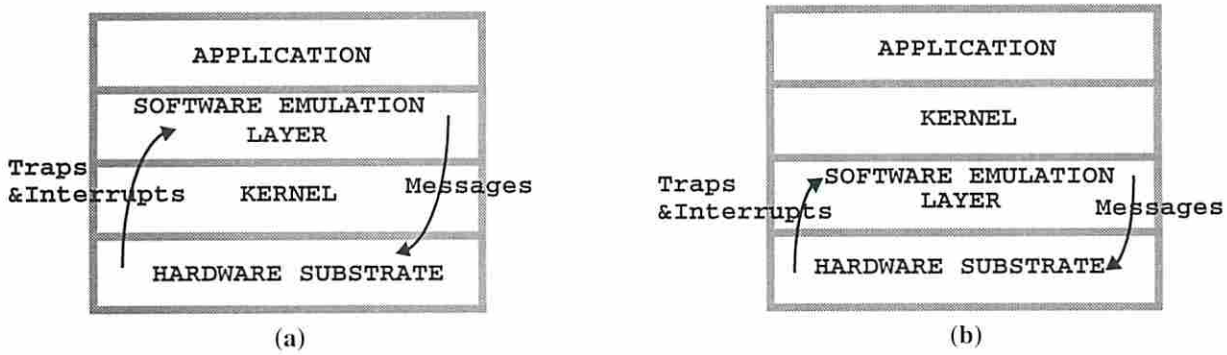

Figure 1 Virtual Shared Memory (VSM) and Direct Software Emulation (DSE) of shared-memory
In a Virtual Shared-Memory implementation (a) the software emulation layer must cross the kernel layer to reach the hardware substrate. In Direct Software Emulation (b), the software emulation layer is adjacent to the hardware substrate.

2.4. CC-NUMA versus COMA

Since our hardware substrate is an interconnected set of independent nodes, no remote memory access facilities are provided and the nodes have to communicate by exchanging messages. In this context, implementing the coherence protocol in software at the level of the caches, as is done in a CC-NUMA machine, requires that the processor can access the cache tags, states, and memory from software trap handlers. Such accesses are currently not implemented in commodity processors and demand a specific design of the caches that significantly increases their hardware requirements [6]. Using the local memory as a cache, as is done in a COMA, is much more natural since it limits the hardware development to the memory controller and can work with standard caches and processors.

From a performance point of view, a COMA has another known advantage, as pointed out in [24]. By using large attraction memories [11], a COMA cuts the number of replacement misses and hence the number of remote memory accesses. This property is very important in the context of a software protocol where the miss latencies are long and data set sizes must be large.

3. HARDWARE SUBSTRATE FOR SC-COMA

To fully understand the hardware requirements and the software implications of the SC-COMA architecture, we are currently developing the hardware substrate of SC-COMA on the RPM emulator. RPM (Rapid Prototyping engine for Multiprocessors) [1] is a flexible hardware multiprocessor platform with eight processors, each attached to a hierarchy of caches and a partition of the memory. The caches and memory controllers are implemented with FPGAs (Field-Programmable Gate Arrays). By programming these controllers, prototypes of message-passing or shared-memory systems can be developed in a short time. From the software point of view, the RPM interface is indistinguishable from a (slow) hardware implementation of a SPARC-based multiprocessor. Each pclock in an instruction is executed in eight clocks of RPM, as if the implementation was microcoded. A non-intrusive event counting mechanism collects performance statistics on the fly. The relative speed of processors, memories and interconnect can be changed easily to emulate various technological environments under a methodology called *time scaling* in [8].

The development of our SC-COMA on RPM includes two phases. First we must prototype the expected hardware substrate and second, we must implement the software trap handlers of the software emulation layer. These two activities have progressed in parallel. In this section we briefly describe the hardware substrate which we are prototyping on RPM. In Sections 4 to 6 we will cover the software handlers.

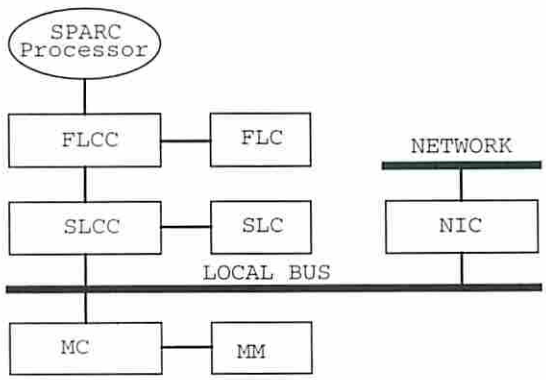

Figure 2 Block diagram of each processing element of SC-COMA

The architecture of each node is shown in Figure 2. Each node is made of a commodity processor and its associated caches. The node also has some private memory used to store local data and code for the application and the trap handlers. As opposed to other studies, we do not assume fast context-switching processors or any other kind of hardware optimization. The architecture could accommodate multiple processors per node. This would, however, require a modification of the implementation and we do not consider this issue in the remainder of this paper.

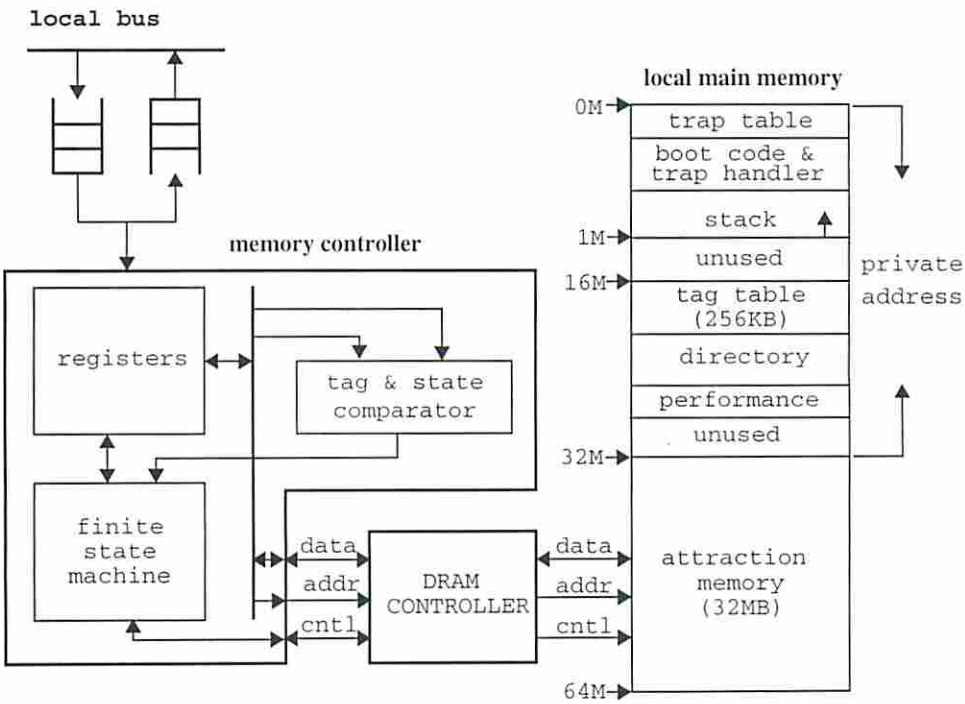

Figure 3 Block diagram of memory controller and memory mapping

Because of the COMA architecture, caches and main memory maintain full inclusion. Consequently, replacements from the caches cannot generate a memory exception and are handled transparently by the hardware. To maintain consistency, cache lines may have to be flushed and/or invalidated when a miss request is served by a node. For this purpose we require two specific instructions, Flush and Invalidate. The Flush instruction generates a write-back of a modified line into the local memory and sets the state of the line in the cache to Shared (Read-Only). The Invalidate instruction simply sets a cache line state to Invalid. Such instructions are commonly implemented in modern processors and do not constitute a real hardware overhead. In the case of

the SPARC processor, these instructions are implemented with Alternate Space Identifiers (ASI). Cache block size and cache size are variable. The network interface is very simple. It contains one incoming and one outgoing memory-mapped message buffer. The processor may send a message by writing into the outbound buffer and the network interface generates an interrupt to the processor as soon as it receives a message.

The main memory on each board (64 Mbytes of DRAM) will contain the attraction memory. The memory controller is connected to the local bus via a bidirectional FIFO and mainly consists of a finite state machine, four word registers for message, address and tag, and a word comparator. The attraction memory is 32 Mbytes with a set-associative organization with four states per block. Besides the attraction memory, the memory contains a tag table, a directory area and private data area (see Figure 3). Each Tag Table entry packs the tags and the state bits for one entire set, so that the memory controller can fetch and compare the tags and state bits of a set in parallel. The directory area is accessible by the software handlers only. Figure 3 shows an area of memory reserved for performance counters used by the count memory, which is the primary performance collection mechanism in RPM.

When the memory controller receives a request (for example *read_miss_request*) from the second-level cache controller, it detects whether the address is for the private memory space or the attraction memory. If it is for the attraction memory, the controller fetches one tag-table entry with the least significant bits of the line address and compares the tags of the whole set with the tag of the processor address. In the case of a match with a valid state, the block is fetched and sent to the second-level cache controller. Otherwise, a memory exception traps the processor[1].

4. SOFTWARE HANDLERS

This paragraph describes the trap handlers in the software emulation layer of SC-COMA. The description is for a Sparc V7 processor (the processor in RPM), but this implementation could be

1. In SC-COMA, we have implemented the detection mechanism in the memory controller because we can easily re-program the controller of RPM. If this is not possible, the tag table should be implemented as a separate unit which snoops on the memory bus, as proposed for the S-COMA on S3.MP [22][19].

ported on most current processors.

A software COMA protocol requires two basic mechanisms. The first one is the ability to trap the processor on a miss in the local attraction memory so that a protocol action can be invoked to bring the desired data in the correct state into the memory. The second mechanism is the ability, for a node, to interrupt a remote node, so that requests and replies can be exchanged.

In the current design of SC-COMA, two types of trap handlers are implemented: synchronous and asynchronous. A synchronous memory exception handler traps the processor 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 in the attraction memory in the correct state. Thus, this trap must be a *precise exception*. In the SPARC architecture it is triggered by the MEXC signal. Another requirement is that a write on a clean block in the cache is detected so that a signal reaches the memory controller. This feature is common in today's high-performance processors whose second level cache includes support for multiprocessor architectures with MOSEI states. A similar assumption is made by Blizzard-E [12], Typhoon [7], Sun S3.mp [19] and the Standford FLASH architecture [16]. Asynchronous interruptions handle remote messages containing shared memory line requests and replies issued by remote processors. At the time of a remote interruption, the processor can be either in the application or in a memory exception waiting for a reply. Asynchronous interruptions are triggered by the IRL signal on the SPARC processor. This signal can encode 16 levels of interruptions. In our design we only use one level.

All the trap handlers developed for SC-COMA to implement the coherence protocol are written in C for simplicity and ease of modification. However, each trap is composed of three parts, a prologue, a C handler and an epilogue. The prologue and the epilogue are written in assembly language. Their function is to save and restore the processor state so that an interruption remains transparent to the current application. The prologue is also in charge of calling the C handler for the corresponding trap (synchronous memory exception or asynchronous interruption). This code is highly optimized, adding an overhead of just 25-30 instructions to the C routines. The context

of the application at the time of the exception is saved entirely in processor registers (trap window locals) most of the time, by making use of SPARC's windowed registers and noticing that the C routines use just one global register. Since the code for the prologue and the epilogue is very likely to hit in the first-level cache and there are no data transfers involved, the transition to the C handlers and back happens in less than 30 pclocks on the average. The C handler implements the coherence protocol actions.

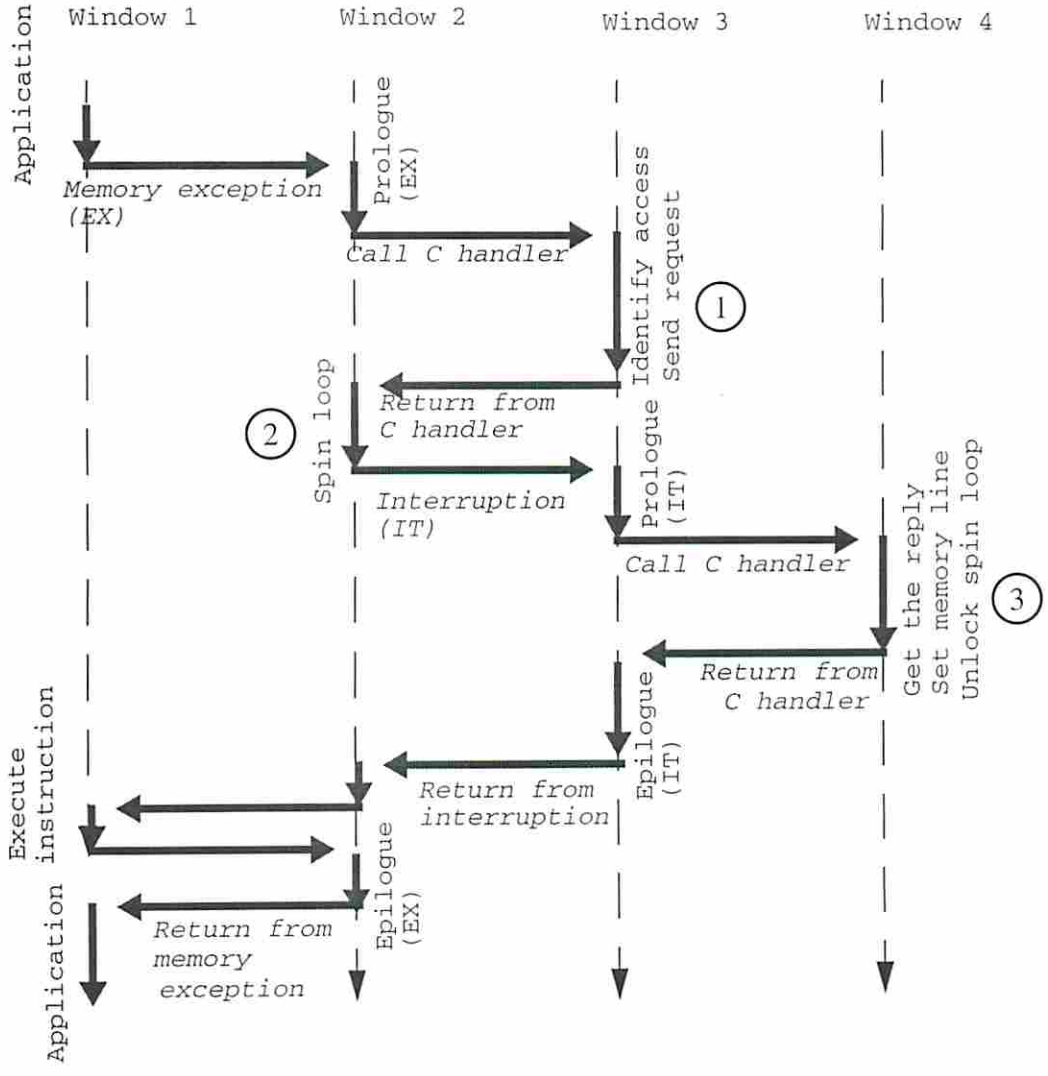

Figure 4 Anatomy of a memory exception

4.1. Memory exception handler

The memory exception handler is called as soon as the trap prologue has been executed. It is composed of three parts. In the first part, the processor identifies the missing line, its current state in

the memory, and the type of memory access (Read, Write, LoadStoreAtomic) by reading a set of memory-mapped registers filled by the memory controller. Based on this information, the processor builds a request which is sent to one of the remote nodes. This terminates the first part of the memory exception handler. The processor then 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. The processor can now restart its computation and re-execute the missing instruction. The different phases of a memory exception are depicted in Figure 4.

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 can be raised and may 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 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 [15]. To avoid livelocks and ensure forward progress, this window of vulnerability must be closed.

One approach for closing the window of vulnerability, which was proposed for hardware coherence protocols, is called *associative locking* in [6]. Associative locking 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 in the memory, the interrupting request must be rejected so that the local processor can restart the access on the line before loosing the line. Adapting such a solution to our implementation would require a new *locked* state for the lines of the attraction memory as well as a modification of the memory controller implementation. To minimize hardware complexity, we have implemented a software solution in SC-COMA. To close the window of vulnerability the instruction that generated the memory excep-

tion is executed in the memory exception handler, before enabling interruptions. As a result, no other interruption is allowed between the end of the interruption and the restart of the memory exception handler. The missing instruction is copied into the code of the handler in a dedicated instruction slot containing normally a nop operation. The instruction executes in the same register window as the application code. After the epilogue, execution resumes at the instruction which follows the missing instruction[2]. A distinct advantage of our software solution over associative locking is that it cannot create any deadlocks.

4.2. The interruption handler

Interruptions are triggered by the network interface when external messages arrive on a node (requests or replies). 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 memory line state and, if necessary, updates the memory with a copy of the requested line. Finally, the handler unlocks the waiting loop in the memory exception handler. No other interruption is allowed within an interruption handler.

5. PROTOCOL

5.1. Basic description

The protocol of SC-COMA is a write-invalidate protocol very much like the one in COMA-F [13]. Each memory line can be in one of the four following stable states: *Exclusive, Master-Shared, Shared* and *Invalid*. These stable states are maintained in the attraction memory and are used by the memory controller to check whether an access on a line is allowed. The *Exclusive* and *Master-Shared* states identify the owner of a memory line. The *Master-Shared* state denotes a read-only copy. The owner of a line is responsible for answering read and write miss requests for the line. It must also ensure that at least one copy of the line exists in the system. Thus, before replacing the line, it must *inject* it on another node.

2. Similar solutions which avoid copying the instruction at a slightly higher cost are possible.

To identify the owner of a line, a node called the home node is statically associated with each line, as in COMA-F, and holds the *owner identification pointer* for the line. On a miss in an attraction memory, the request is sent to home. If home is also the current owner, it directly replies to the request. If not, it forwards the request to the current owner.

As opposed to COMA-F, the line owner also maintains the current *copyset* (presence bits) for the line. Associating the copyset to the current owner of the line (instead of home) allows the owner to send the invalidations without consulting the home node. In the current implementation, the protocol implements a sequential consistent memory access model with no prefetching; hence only one request can be pending in each node at a time.

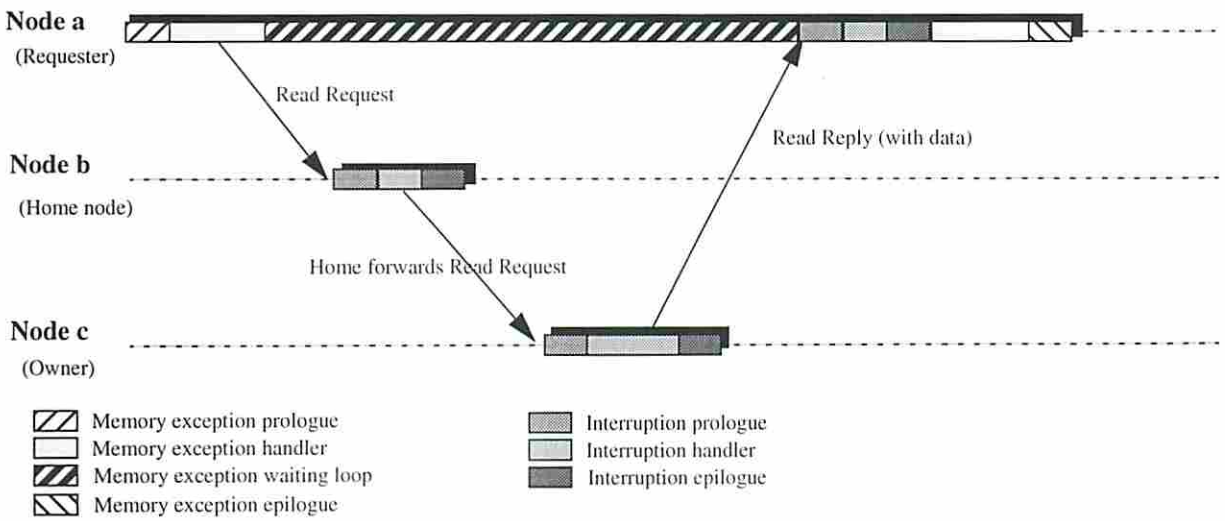

Figure 5 Timing diagram for a remote read miss

After detecting a miss in the memory, a memory exception is triggered. A Read Request message is sent to the home node by the exception handler, which then enters a busy waiting loop. The home node is interrupted by the message and forwards the request to the current owner of the line. The interruption handler at the owner sends a Read Reply message to the requesting node, flushes its local caches, and changes the state of the line in its memory to Master-Shared if the line was previously Exclusive. The owner also adds the requesting node to the copy-set of the line. When the reply finally reaches the requesting node, the line is copied to the memory, its state is updated and the waiting loop of the memory exception handler is released. The requesting processor then comes back in the memory exception handler and executes the missing instruction before returning to the context of the application

5.2. Implementation

When implementing a software coherence protocol, it is important to minimize the number of messages that the nodes have to exchange, the number of node interruptions, and finally, the time

of each interruption. To achieve this goal, we use the flexibility offered by a software implementation. While a node is waiting for ownership of the line, the home node points to this future owner. Instead of Nacking incoming miss requests, the future owner keeps these requests in a software buffer. Once the transaction is completed, all the buffered requests are answered.

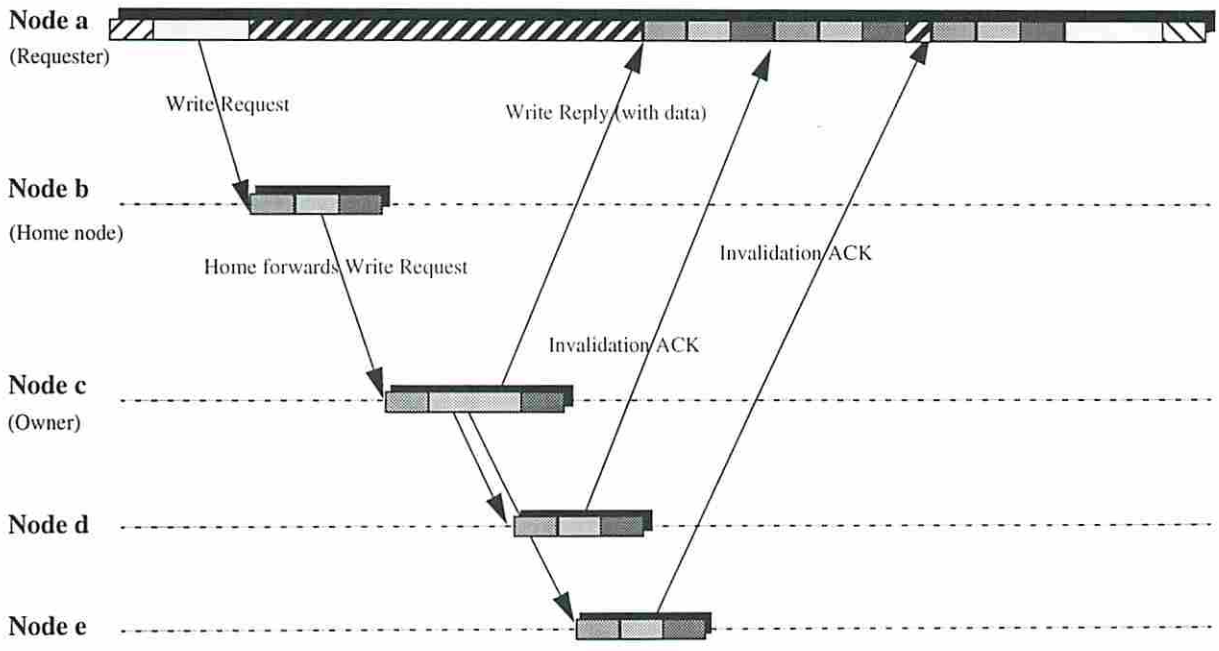

Figure 6 Timing diagram for a remote write miss request

In the implementation, we have tried to minimize the interruption time of the nodes. Invalidation acknowledgments are collected by the future owner of a line and not by the current owner. When it receives a write request, the owner of a line performs the following actions: (1) it forwards a copy of the line and the current copyset to the requesting node (write reply message), (2) it checks the copyset and sends an invalidation to each node in the copyset. At this point the node loses ownership and its interruption handler is terminated. A node receiving an invalidation message simply invalidates its copy of the line (in memory and in the caches) and sends an invalidation acknowledgment to the future owner of the line. As soon as the requesting node receives the write reply and all the invalidation acknowledgments, it can restart its computation.

This strategy limits the number of messages exchanged by the nodes as well as the number of interruptions. In the worst case, a read request is completed in three messages, three interruptions and one memory exception. Since the modification of the owner identification pointer at the home node is done as soon as a write request is forwarded, one message and one interruption to update the owner identification pointer at the end of a transaction are saved as compared to the COMA-F protocol. In order to limit the number of messages, a node can replace a clean line without updating its copyset. As a result the copyset maintained by the owner of a line is a superset of

the real set of nodes having a shared copy of a line.

Figure 5 depicts the phases of a remote read miss request. As shown in Figure 6 for the case of a write request on a shared line, the phases are the same except that the requester has to wait for all the invalidation acknowledgments before the request is completed. Note that invalidation acknowledgments are collected in the requester node. This optimization save cycles since no other node is interrupted to collect invalidation acknowledgments and the requester has to spin in any case. .

5.3. Replacement in the Attraction Memory

Since there is no physical memory to back up the attraction memories, the protocol must avoid replacing the last copy of a line. Moreover, to prevent deadlock situations, the amount of shared memory allocated should always be less than the sum of all attraction memories. A replacement is triggered when a miss occurs in an attraction memory and no other free memory line can be allocated in this memory. The protocol chooses a victim line according to the following priorities: (1) Invalid, (2) Shared, (3) Exclusive or Master-Shared. Replacing an Invalid or a Shared line is harmless. To replace an Exclusive or a Master-Shared line, a replacement request carrying a copy of the line is always sent to the home node for injection in a different node. If the home does not accept the injection, the request is forwarded from node to node until it finally finds its place. A injected line can only replace an Invalid or a Shared line and does not generate other replacements in the visited attraction memories. During this injection, the pointer for the line is locked at the home node and any request for the line is Nacked. Replacements is the only situation where the protocol uses Nacks.

5.4. Data structures used in the implementation

Figure 7 shows the logical organization of an attraction memory. The Pointer Table and the Copy-set Table form the directory and are only accessed by software handlers in the local processor. Thus they are cached. The Pointer Table contains the owner identification pointers of all the lines

for which the node is home. The distribution of the owner identification pointers on the nodes is defined by software. In our implementation, the pointers are distributed across the nodes according to the low order bits of the line addresses. Thus there are as many pointers as there are lines in a node. The size of a pointer is the logarithm of the number of nodes in the architecture. A copyset contains the presence bits for a line cached in the attraction memory of the node as an owner. The size of a copyset is equal to the number of processors and we need as many copysets as there are lines in the local attraction memory (since every line could be owned).

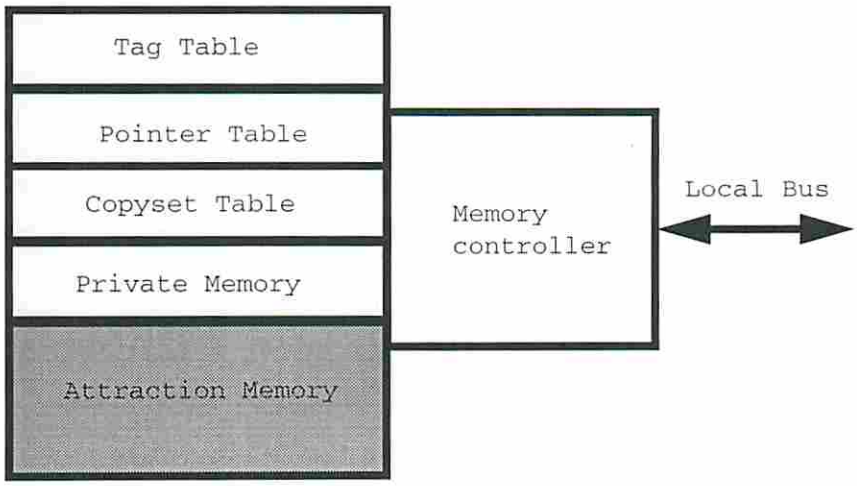

Figure 7 Logical description of an attraction memory

The Tag Table maintains a tag and two state bits for each line in the attraction memory, and is accessed by both the memory controller and the software handlers. Their main function is to detect a miss in the attraction memory. The Tag Table can be cached, provided the cache is flushed at the end of the handler modifying a tag. In our current simulation, we do not cache entries of the Tag Table. Finally, the private data area is for the private data of the local node, which is not cached in the attraction memory.

6. EVALUATION OF SC-COMA

In order to design the protocol and debug the software handlers independently of the hardware prototype, we have developed a detailed simulation of the hardware substrate. The simulator has allowed us to quickly change the protocol, and therefore to understand ways to improve it. The

protocol presented in this paper is the result of multiple iterations. The simulator allows us to compare SC-COMA with a CC-NUMA implementation on the same hardware. Our performance results are based on execution-driven simulation of the SPLASH-2 benchmarks [25].

6.1. The Simulator

The simulator is RPMsim, a flexible simulation environment for the RPM emulator. The common module incorporates a processor simulator, private and shared physical memory modules, a FIFO message-passing substrate (bus or crossbar), two levels of caches, and a custom event scheduler which implements the multiprocessor features. A statistics-gathering module records and categorizes relevant events. A separate module for each architecture implements the cache and memory controllers.

The processor simulator has evolved from CacheMire-2 [3] and is an interpreter for the instruction set of the SPARC architecture. It features trapping in the event of an exception 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 memory reference. Every instruction executes in one processor clock and, because we do not simulate the details of the instruction pipeline, the code for the simulator is highly efficient (1.7 million references per second, on a SPARC-20). The binaries of both the software handlers and the application code are executed completely and faithfully by the simulator.

6.2. Baseline Architecture

The simulated architecture is very similar to the RPM implementation except that number of nodes is 16. Each node contains a 100 Mhz Sparc processor, a hierarchy of caches, and a memory module. The first-level caches (FLC) and second-level caches (SLC) are both direct-mapped and use 64 bytes lines. The FLC is made of two independent caches, one for the instructions (IC) and

one for the data (DC). Each of these caches is 32 Kbytes. The FLC-data cache is write-through with no allocation on write. The SLC is unified and has a size of 128 Kbytes.

The memory on each board is 4-way interleaved with a 16-byte wide interface and has a memory access time of 8 cycles (80 ns), resulting in a global access time of 24 cycles to read a 64-bytes line. The memory controller implements a 4-way set associative attraction memory. The tags and states of the lines are in DRAM and the time necessary to access and check the tags is 13 cycles (8 cycles to access the memory and 5 cycles to compare the tags and send the reply). The memory and the SLC are connected by a 128 bits local bus running at 100 Mhz. The time to transfer a cache line of 64 bytes from the memory to the second level cache is thus 6 cycles.

The network interface is connected to the local bus. Control of the network interface is provided by a set of memory-mapped registers. Copying a message to or from the network interface takes 8 cycles for a message without data (8 bytes) and 136 cycles for a message containing a memory line (136 bytes). When a reply message contains a line, we assume that a DMA transfer is used to copy the line to memory. Copying the line from the network interface to the memory takes 40 cycles. The simulated network is a crossbar network where each path between two nodes provides a constant bandwidth of 100 MB/s between two nodes. Hence the time to transfer a request (8 bytes) from one node to another node is 8 cycles and the time to transfer a message containing a line (136 bytes) is 130 cycles. These interconnection latencies are very short. In the case of an ATM network, the latencies would be considerably longer and therefore the penalties introduced by the software protocol would be relatively much smaller than reported here.

In the trap handler, identifying that a line is present in the memory requires an access to the memory in order to retrieve the tags and states for a particular set in the attraction memory. As such data is not cached, accesses to the Tag Table disable the caches and directly go to the memory. In the simulated implementation, a single double word access (8 bytes) is sufficient to load all the states and tags for a single set of the memory. Such an access is assumed to take 17 cycles

since the caches do not copy the data.

Latencies for Read Requests	Number of cycles (cycle = 10 ns)	
	COMA	CC-NUMA
Read miss served by FLC	1	1
Read miss served by SLC	6	6
Read miss served by the local memory: Private data (64 bytes)	37	37
Read miss served by the local memory: Shared data (64 bytes)	37 + 13 = 50	37
Direct read access from memory (4 bytes, uncached)	17	Na
Read miss in a remote node	Software dependent	99

Table 1: Read miss latencies (no conflict)

The CC-NUMA architecture to which we compare our SC-COMA share similar characteristics. The memory access time is, however, smaller (37 cycles) since no tag comparison is necessary. The coherence unit is also smaller (64 bytes) since coherence is enforced among second-level caches. The transfer of a line between two nodes is thus decreased to 62 cycles (instead of 130 cycles).

The data set sizes that we could simulate are still very small. To account for this effect and observe some replacements in the attraction memory of SC-COMA, we have cut the amount of attraction memory in each node to 4 Mbytes. When the RPM emulator is running we will be able to look at data set sizes which fill the whole 32 Mbytes attraction memory on each node.

6.3. The Benchmarks

The SPLASH-2 benchmarks are compiled with gcc-2.7.0 -O2 and linked with the system libraries of SunOS 4.1.3. A special library provides routines required by the ANL macros used in these applications. The implementation of the locking primitive consists of a load-store atomic followed by busy testing in case of failure. Pauses make use of these shared-memory locks. Barrier synchronization is performed in the simulator, using a special software trap, to reduce the simula-

tion time. Other services, such as G_MALLOC, CREATE, CLOCK, employ the same trapping mechanism. The trap table and software handlers for the COMA are linked together with the application to create the executable which can be used for both COMA and CC-NUMA simulations.

Benchmark	Parameters	Shared memory used (bytes)	Memory pressure
Barnes	4K particles	8,592,068	12.8%
Cholesky	tk15.O	18,913,924	28.2%
FFT	64K points	3,359,104	5.0%
LU	256 x 256	528,260	0.8%
Ocean	258 x 258	15,743,632	23.5%
Radix	1M integers	8,586,492	12.8%
Volrend	head-scaleddown2	2,755,587	4.1%
Water-Nsq	512 molecules	4,587,924	6.8%

Table 2: Characteristics of the Benchmarks

The characteristics of the benchmarks are given in Table 2. The memory pressure is the ratio of the attraction memory space used by the application and the sum of the attraction memory sizes (64 Mbytes).

6.4. Simulation Results

6.4.1. Miss Latencies

To optimize the performance of the SC-COMA and to compare it with other implementations, it is important to fully understand where the time is spent during a miss. To analyze these times, we have run a set of simple programs (micro-benchmarks), each of which targets a particular situation such as a read miss or a write miss with one or multiple invalidations. For each of these programs, we collect the time spent in the different trap handlers and add the time spent in the network. In Figure 8, each stacked bar gives the time breakdown in cycles for the latency of a particular 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 execute the missing instruction when the reply is received. For write misses with invalidations, the invalidation acknowledgment handler time represents the sum of all the interruptions (one for each invalidation sent).

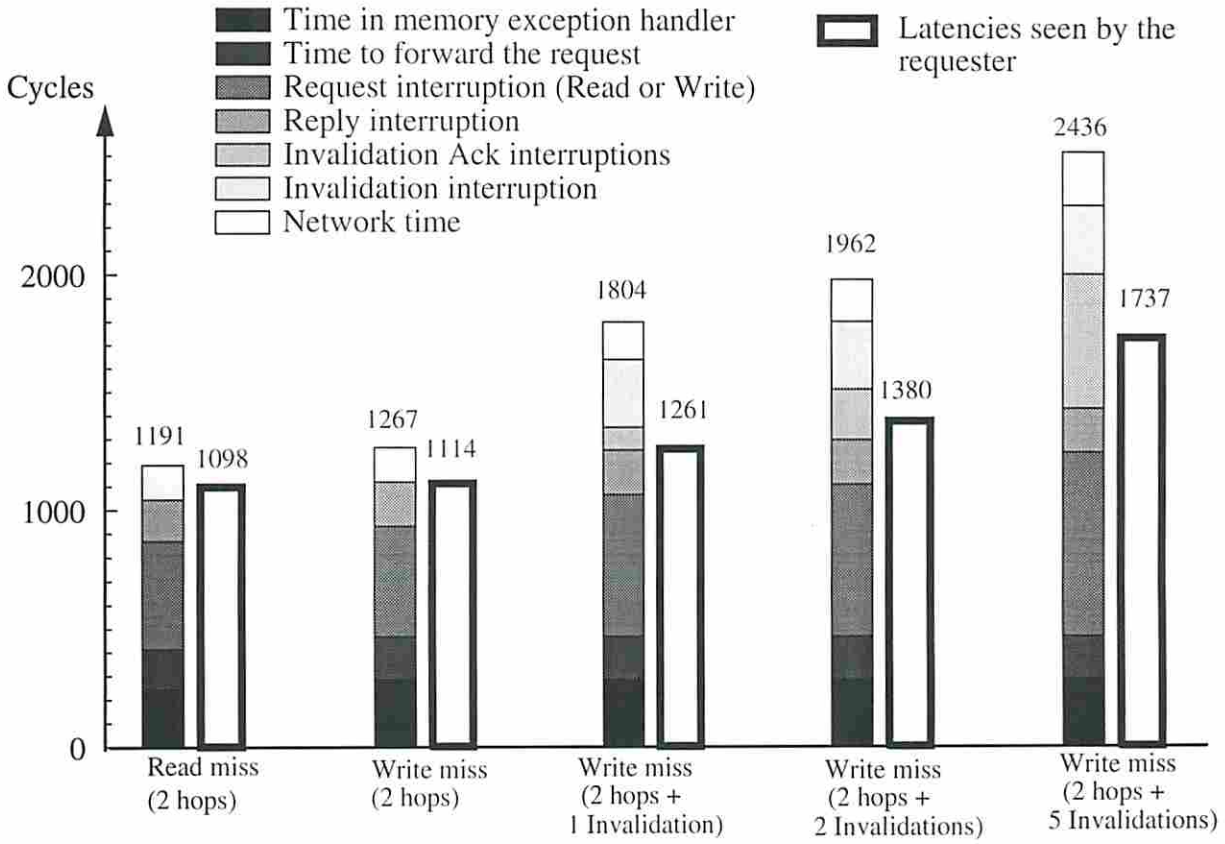

Figure 8 Breakdown of latencies for simple requests (micro-benchmarks)

The bar on the right of each stacked bar gives the latency experienced by the requester. The difference between the execution time and the sum of the different timings of a request is due to the fact that some of these timings overlap. For example, in a read request handler, the reply message containing the data is sent as soon as possible and the network transfer overlaps with the execution of the end of the handler. We now describe the components of the latencies.

First of all, forwarding a request to the current owner of a line takes 168 cycles for a read request and 182 cycles for a write request because the home processor has to update the owner identification pointer of the line.

Second, the most important component of the latency is the remote read or write interruption handler at the owner. Before being able to send the reply request, the owner must identify where the line is in memory. This requires fetching the tags of the lines of the set from memory, and comparing the tags in the Tag Table with the address tag of the request. For either a read or a write request, this time represents 460 cycles. The comparison could be speeded up if the software could use the tag comparison hardware in the memory controller or if the degree of associativity of the attraction memory was reduced. For write requests with invalidations, the time to send the invalidations has to be added to this value. The first invalidation sent increases the handler time by 140 cycles, then each new invalidation sent by the owner adds an overhead of 45 cycles.

Third, invalidations must be received and acknowledged in the case of a write request. On the graph, we only show the time for one invalidation interruption, since all the invalidations handlers are executed in parallel. On a node with a Shared copy of a line, the invalidation interruption time is 290 cycles. However, a big part of this latency is hidden since the invalidation acknowledgment message is sent before searching for the line and invalidating it. Invalidation acknowledgments are directly sent to the node requesting the line. Receiving an invalidation acknowledgment consumes approximately 100 cycles.

Looking at the actual times, we see that the global cost per invalidation is close to 120 cycles. This can represent a large overhead if data sharing is important. It has, however, been shown that 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 is advocated in [4] would yield a large performance improvement, except for benchmarks with wide sharing. This issue is also discussed further in the next section.

6.4.2. Execution Times

In this section we present the breakdown of the execution time of each application and compare the execution time on the SC-COMA with the execution time on the CC-NUMA with the same hardware parameters. Both systems are sequentially consistent. For each application two bars are shown. The left bar gives the execution time on the CC-NUMA architecture. This time is divided into three parts: busy, stall, and barrier synchronization. The right bar gives the execution time on SC-COMA decomposed into seven components: busy, local stall, barrier synchronization, time spent in memory exception handler, spin (time spent in the waiting loop of the memory exception handler), time interrupted in application, and time interrupted in memory exception handlers. The time spent in interruption handlers during synchronization is not counted in the time interrupted in the application.

By looking at the graph of Figure 9, we first observe that the software protocol of SC-COMA is a viable approach. Barnes, Cholesky, Volrend and Water all show busy times superior to 50%. Second, the amount of time spent interrupted, either in the application or in a memory exception by other processor requests, remains under 15% for most the applications which seems to imply that using a communication coprocessor would not dramatically improve performance. Out of the eight applications that we have run, only Radix and Fft spend more than 15% of their time in interruptions. Both of these applications exhibit high miss ratios due to true sharing (coherence misses), which explains why they do not behave well on SC-COMA [13].

In Fft we observe an unusual slowdown by a factor of almost four in the second transposition phase. This is due to an uneven distribution of the coherence misses among the processing nodes. The resulting contention overloads some of the nodes, especially node 0. As a result, node 0 is interrupted three times more frequently than the other nodes and spends more than 56% of its time in interruptions. Because of the contention created on node 0, the average read and write

miss times increase to almost 3000 cycles and all the other nodes spend an important part of their time spinning and waiting at barrier synchronizations.

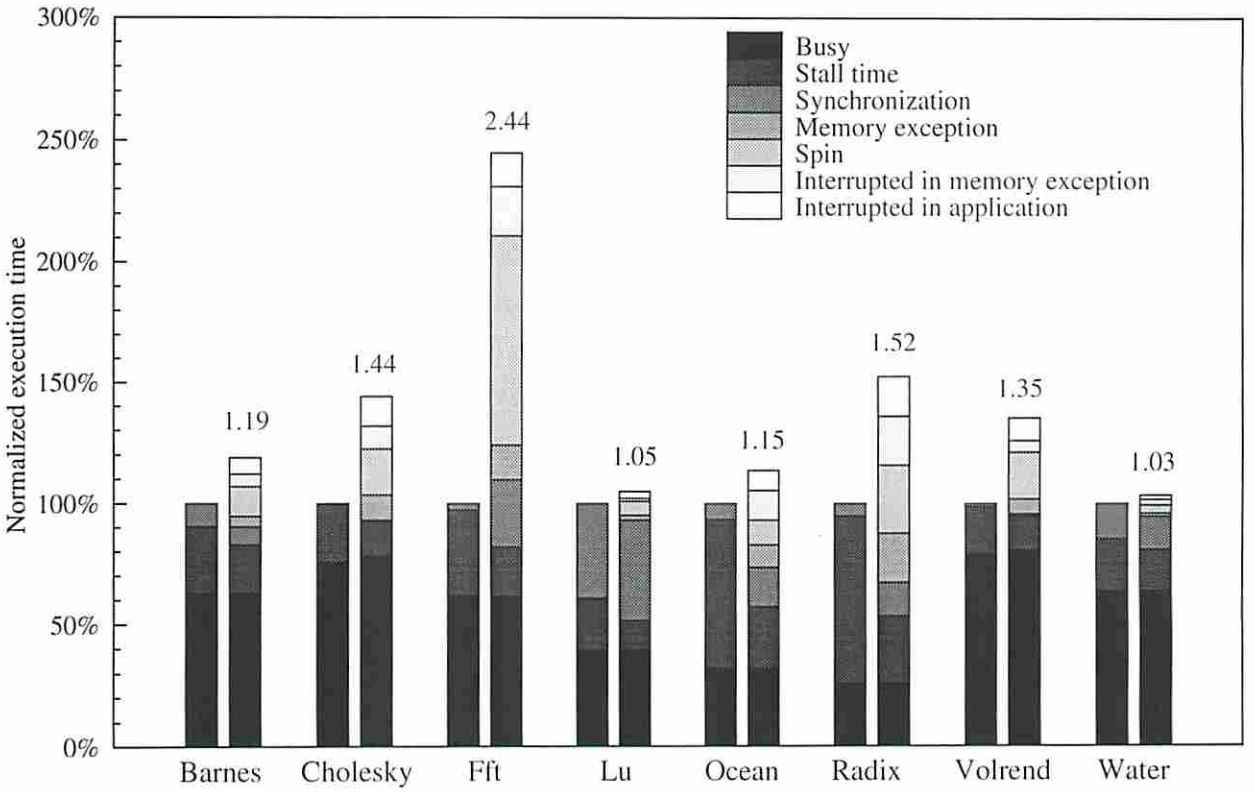

Figure 9 Normalized execution time

The left bar shows the three components in the execution time on the CC-NUMA, normalized to the total execution time. The right bar is the breakdown of the execution time of SC-COMA normalized to the execution time of CC-NUMA. The number on top of the bar is the slowdown of SC-COMA.

By comparing the execution times obtained with the CC-NUMA architecture and with SC-COMA, we observe that all the times are increased in SC-COMA. The increase ranges from more than 2 times for Fft to 1.04 for Lu. Applications like Barnes, Lu and Water benefit from the attraction memory, which drastically reduces the number of remote requests generated by the nodes. As a result the long miss latencies of the SC-COMA are practically hidden by the large number of misses in the CC-NUMA architecture. As an example, the number of memory misses of Water in SC-COMA is 14.7 times smaller than the number of SLC misses in the CC-NUMA architecture. Cholesky, Ocean, Radix and Volrend have a large number of true sharing misses that

can not be reduced by the size of the attraction memory (the ratio of the number of misses in the SLC of the CC-NUMA architecture and in the memory of SC-COMA are respectively: 4.9, 11.5, 11.8, and 5.1). For these applications, the longer miss latencies of SC-COMA result in longer execution times. Note however, that even for these applications, the ratio of execution times in the two architectures is still acceptable.

6.5. Possible optimizations

Different optimizations, either hardware or software, could be applied in order to reduce the memory exception times. In this section we present a few of them and, based on our experience, try to identify what would be their impact on our current implementation.

6.5.1. Hardware optimizations

Saving the processor state at each interruption can be a costly operation. Even with a highly optimized code, we measured that the time to save and restore the processor context is close to 30 cycles. Processors such as UltraSparc or MC88100 which provide a set of fresh registers for trap handlers and hence do not require saving the registers on interrupts could remove a part of these cycles. Considering that a read miss memory exception takes about 1,300 cycles (with conflicts) and uses 4 interruptions, at most 120 cycles (about 10%) could be saved in the miss latency. Another effect of this optimization would be to decrease the interruption time on remote nodes, hence possibly increasing their busy time.

Dispatching invalidations and collecting invalidation acknowledgments in hardware is an optimization proposed in [4] and advocated as a key mechanism for software coherence protocols. The effect of this optimization is that it could possibly reduce the write miss times by speeding up the dispatch of invalidations and by removing the interruptions necessary to receive the invalidation acknowledgments. Based on the results we obtained with the applications considered in this

study we do not think that hardware support would make a big difference by itself (it may cut the miss latency by up to 10%). The applications do not show wide sharing of data, resulting in a low number of invalidation messages. Actually, we think that only actively shared data structures, such as synchronization locks, could really benefit from this optimization.

Active messages or multiple level interruptions are other solutions that could reduce the overheads of traps. Both can speedup the selection of the handler for a request by removing the time necessary to decode the received message. Some of the interruptions, such as the interruptions forwarding a request at the home, could then be highly optimized.

6.5.2. Software optimizations

Numerous software optimizations can also be applied to SC-COMA. As in VSM systems, SC-COMA could adopt a more relaxed consistency model to cut the number of communications between nodes and to hide the latency of stores. The implication of using more relaxed consistency models will be a greater complexity of the handlers code to take into account several possible pending requests. This should result in an increase of the latencies of memory read and write misses. Whether the gain provided by a relaxed consistency model is worthwhile is an open question, which would require another study.

Another software optimization concerns synchronization locks. Locks are accessed by all nodes and can be the source of an important performance degradation because they are actively shared. Implementing queue-based locks should be rather straight forward in our current implementation since the data structure to buffer a request is already implemented. Such locks will reduce the number of misses caused by the test and test&set algorithm in the current implementation.

Writing the handlers in assembly code instead of C would not provide a significant

speedup and would affect the flexibility of the protocol. This flexibility is potentially a rich source of improvements since new protocol mechanisms can be easily added in software controlled protocols.

7. COMPARISON WITH OTHER PROJECTS

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 [5]. However, recent work has concentrated on system-level approaches. A major difference among approaches is whether the hardware substrate is shared-memory (NCC-NUMA) or distributed (message-passing).

The first paper to trigger interest in system-level approaches on top of a NCC-NUMA hardware substrate was related to the Alewife project [4]. The paper compared a DSE of the Alewife protocol with various software-assisted directory-based implementations of a CC-NUMA. The paper only compared total execution times in the context of a sequentially-consistent architecture. The software-only solution was surprisingly competitive with the software-assisted solution, especially considering the very small data set sizes. The paper recommended that a hardware mechanism be present to send and collect invalidations. However, our data on the SC-COMA show that, whereas sending invalidations and receiving their acknowledgments in software is a time-consuming process, it is not the most important one.

On the heels of the Alewife study, Hakan Grahn and Per Stenström evaluated various implementations of a software-only protocol for a CC-NUMA on top of an NCC_NUMA substrate [10]. All their solutions assume a hardware mechanism for the dispatch of invalidations and the collection of invalidation acknowledgments. Besides the NCC-NUMA hardware substrate the architecture also requires lock-up free caches in order to prevent deadlocks. Software is only used on the home node for the management of the directory. Finally, the performance reported in the paper makes strong assumptions on the hardware implementation and hence does not provide

strong information about software-controlled implementations of a coherence protocol.

Another project to emulate a CC-NUMA on top of a NCC-NUMA substrate is Karen Peterson and Kai Li's work [20] on virtual memory supported cache-coherence, later perfected by Kontothanassis and Scott [14]. 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.

Other approaches have tried to extend the features provided by the virtual memory system on a distributed-memory (message-passing) substrate, following Kai Li's pioneering work on virtual shared memory [18] [17]. All these systems have a COMA-like flavor. Blizzard-E, for example, is a VSM system that uses the CM5 virtual memory system to allocate memory and the CM5 memory error-correcting codes (ECC) to implement a memory line valid bit and support fine grain sharing. The software overhead of traps and messages through the kernel have been reduced by careful recoding. The latency of a remote miss is still of the order of 6,000 cycles (as compared to 1,000 cycles for SC-COMA). Depending on the application, Blizzard-E is slower than KSR-1 by a factor 2 to 10 [23]. Instead of optimizing the kernel code and implementing fine-grain sharing extensions, Tread Marks [9] relies heavily on relaxed memory models to overcome the very large latencies of operating system calls. Latencies are in the 10's of thousands cycles. Multiple-writer protocols with delayed writes, combined in software diffs and sent at *acquires*, cut down on the large traffic and the large miss rate caused by false sharing [7].

The simple COMA [22] currently advocates exploiting the virtual memory support in the kernel in conjunction with a protocol engine connected to the communication network keeping track of the coherence information for each line of the local memory. In this architecture, the MMU performs the attraction memory tag check at the page level, and the protocol engine checks to see if the accessed line is valid in the memory and implements the coherence actions to retrieve

the line. As in these systems, physical addresses point to the location of data in local memories, they must be translated to and from global addresses that can be used as global identifiers.

Finally, the last approach consists of using a dedicated reprogrammable processor to implement the coherence protocol [16], [19], [21]. Such a solution keeps the flexibility of a full-software approach, as well as some of the efficiency of a hardware approach, because the computation processor is not interrupted. Communication may proceed in parallel with computations and the protocol processor can be optimized for protocol execution. Miss latencies are in the order of 100's of cycles. The overall cost of such a system is much higher, however, than a standard VSM system and, in final analysis, may not be cost-effective.

8. CONCLUSION

In this paper, we have described the design of a software-controlled COMA on a message passing hardware substrate based on direct software emulation of the protocol on the processor. The software protocol is sequentially consistent and invalidation-based. Software actions are triggered by low-level synchronous and asynchronous interruptions. Commodity processors with on-chip caches are assumed. The only hardware support is (1) hardware support in the second-level cache for detecting writes on clean blocks, (2) a mechanism at the memory to detect the presence of a line and trap the processor if needed (as an alternative this function could be fulfilled by a snooping device on the memory bus), and (3) a precise memory exception in the processor. We do not require any special feature such as fast context switching processors or hardware support for the dispatch of invalidations and the reception of invalidation acknowledgments.

Because we do not rely on operating system services, latencies of traps and messages are drastically cut with respect to traditional VSM systems. Moreover, we maintain fine-grain coherence on cache blocks. A read miss on clean remote takes about 1,000 processor cycles on SC-COMA. This compares with latencies of 10's of cycles for snooping protocols, 100's of cycles for directory-based hardware protocols and 10's of thousands cycles for VSM systems. Overall SC-COMA is competitive with hardware CC-NUMA and yields good processor efficiency for most

of our benchmarks.

The protocol has been designed to minimize the effects of some well-known weaknesses of software-controlled protocols such as the cost of processor context switching and the handling of invalidations. Future processors may have a larger latency than we have assumed for flushing their pipeline on precise exceptions, but they may also have additional sets of registers for fast trap handling (which we do not assume). By optimizing the prologue and the epilogue of each trap handler, we have cut the overhead of context switching to less than 10% of the miss latency time. Similarly, by collecting invalidation acknowledgments in the requester node (which must spin in any case), we have hidden the cost of these traps as well. The results of our evaluations show that no one particular hardware or software improvement could dramatically enhance the performance of SC-COMA as presently designed. Nevertheless, improvements are possible and several optimizations have been discussed in the paper. A rich prospect for improvements is to take advantage of the flexibility of Direct Software Emulation to implement more complex protocols. The increased complexity of the handlers may, however, nullify any improvement derived from additional protocol mechanisms.

9. REFERENCES

[1] L. Barroso, et al. RPM: A Rapid Prototyping Engine for Multiprocessor Systems. *IEEE Computer*, pp. 26-34, February 1995.

[2] M. Blumrich et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. *Proc. of the 21st Int. Symp. on Computer Architecture,* pp. 142-153, 1994.

[3] M. Brorrson, F. Dahlgren, H. Nilsson, and P. Stenström. The Cache-Mire Test Bench--A Flexible and Effective Approach for Simulation of Multiprocessors. *Proc. of the 26th Ann. Simulation Symp.*, pp. 41-49, 1993.

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

[6] D.R. Cheriton, G.A. Slavenburg and P.D. Boyle. Software-Controlled Caches in the VMP Multiprocessor. *Proc. of the 13th Annual International Symposium on Computer Architecture.* Tokyo, Japan, June 1986.

[7] 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.

[8] M. Dubois, A. Gefflaut, J. Jeong, A. Moga, and K. Oner. Multiprocessor Emulation with RPM: Early Experience. *submitted to ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, October 1995.

[9] 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.

[10] 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.

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

[12] K. Hwang. Advanced Computer Architecture: Parallelism, Scalability, Programmability. Mc Graw-Hill, 1993.

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

[14] L.I. Kontothanassis, M.L. Scott. Software Coherence for Large Scales Multiprocessors. *Proc. of the First Symposium on High-Performance Computer Architecture*, pages 286-295, Raleigh, North Carolina, January 1995.

[15] 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.

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

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

[18] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Computer Systems*. 7(4), 321-359, Nov. 1989.

[19] 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.

[20] K. Petersen and K. Li. Multiprocessor Cache Coherence Based on Virtual Memory Support. *Journal of Parallel and Distributed Computing*, Vol. 29, No. 2, pp. 158-178, September 1995.

[21] S.K. Reinhardt, B. 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] 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, Raleigh, January 1995.

[23] 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.

[24] 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.

[25] 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.