Memory Organizations in Hybrid DSM: A Performance Comparison

Adrian Moga, Alain Gefflaut, and Michel Dubois

CENG 97-02

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

January 1997

Memory Organizations in Hybrid DSM: A Performance Comparison

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 97-02

Abstract

Hybrid Distributed Shared Memory (DSM) systems are shared-memory multiprocessor architectures with software-implemented coherence protocols and hardware support for fine-grain sharing. In this paper we compare the designs of three possible hybrid architectures inspired from three hardware DSMs: CC-NUMA, COMA and Simple COMA. The corresponding hybrids are called SCC-NUMA, SC-COMA and SS-COMA. There are several competing trade-offs in these three architectures, which require careful evaluations. We have completely implemented the software protocol handlers for the three architectures. In the target systems, these handlers run on the same processor as the application; however, our findings can be easily interpreted in the context of machines with dedicated protocol processors. We base our evaluations on detailed execution-driven simulations of six complete benchmarks with both coarse-grain and fine-grain sharing.

The major difference between SCC-NUMA and the two hybrid COMAs is that both COMAs support automatic data replication in main memory using a fine-grain coherence unit. The difference between the two COMAs is the allocation granularity in memory at the time of replication. SS-COMA allocates memory in pages, which affects the node hit ratio and causes memory pressure inflation. Thus, whereas its hardware implementation is the simplest of all three architectures, its performance is poor for benchmarks with fine-grain sharing. The performance of SCC-NUMA can be very good provided careful attention to page placement. Finally, SC-COMA, allocates memory at the granularity of the coherence unit and its performance is one of the best for all six benchmarks, even when memory pressure is high, cache conflicts are high, and page mapping is poor.

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

Memory Organizations in Hybrid DSM: A Performance Comparison

Abstract

Hybrid Distributed Shared Memory (DSM) systems are shared-memory multiprocessor architectures with software-implemented coherence protocols and hardware support for fine-grain sharing. In this paper we compare the designs of three possible hybrid architectures inspired from three hardware DSMs: CC-NUMA, COMA and Simple COMA. The corresponding hybrids are called SCC-NUMA, SC-COMA and SS-COMA. There are several competing trade-offs in these three architectures, which require careful evaluations. We have completely implemented the software protocol handlers for the three architectures. In the target systems, these handlers run on the same processor as the application; however, our findings can be easily interpreted in the context of machines with dedicated protocol processors. We base our evaluations on detailed execution-driven simulations of six complete benchmarks with both coarse-grain and fine-grain sharing.

The major difference between SCC-NUMA and the two hybrid COMAs is that both COMAs support automatic data replication in main memory using a fine-grain coherence unit. The difference between the two COMAs is the allocation granularity in memory at the time of replication. SS-COMA allocates memory in pages, which affects the node hit ratio and causes memory pressure inflation. Thus, whereas its hardware implementation is the simplest of all three architectures, its performance is poor for benchmarks with fine-grain sharing. The performance of SCC-NUMA can be very good provided careful attention to page placement. Finally, SC-COMA, allocates memory at the granularity of the coherence unit and its performance is one of the best for all six benchmarks, even when memory pressure is high, cache conflicts are high, and page mapping is poor.

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. In large-scale systems, the shared memory is physically distributed among processing nodes and a hardware, directory-based protocol provides a fine-grain access control to memory [23]. However, hardware implementations are complex to build and cannot be changed or improved easily. Since processor speeds are practically doubling every 18 months, ad-hoc hardware approaches, which lengthen machine development cycle, have recently been under intense scrutiny.

For these reasons, software implementations of shared memory on top of a message-passing hardware substrate have been proposed as an alternative to hardware-coherent shared-memory multiprocessors. Software DSM (Distributed Shared Memory) systems such as Ivy [15], or Treadmarks [7] implement shared memory with commodity hardware and software-only protocols. Shared pages are replicated in the memory of each processing node and the address translation hardware controls accesses to shared pages and detects page misses.

Typically the performance of software DSM systems is worse than hardware DSM systems. The software approach is usually based on a user-level implementation on the host operating system requiring costly system calls to send and receive messages and to access and modify page tables entries. Moreover, 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 [6]. On the plus side, however, software DSM systems are highly flexible and can fight the long remote memory latencies with more complex coherence protocols such as adaptive protocols or very weak memory consistency models. These implementations rely on careful application programming and are mostly suitable for applications with coarse-grain sharing.

To improve the performance of software DSM systems for applications with fine-grain sharing several hybrid hardware/software approaches are currently being explored. In these approaches, some hardware support is provided to assist the software. Some examples include support for remote writes in the SHRIMP project [3], support for remote reads and writes in the Cashmere project [12], and support for fine-grain sharing in Blizzard-E [22], Typhoon-0 [19] and START-NG [5]. This trend goes all the way to adding a dedicated processor to execute the software protocol and interact tightly with the network interface, as in Typhoon [18] [19] or Flash [14].

This paper explores three memory organizations for hybrid DSM systems. These hybrid architectures are inspired from three hardware DSMs: CC-NUMA, COMA [24], and Simple COMA [20] and are respectively called SCC-NUMA (Software CC-NUMA), SC-COMA (Software Controlled COMA) and SS-COMA (Software Simple-COMA). These three hybrid architectures rely on a memory Access Control Device (ACD) which satisfies memory hits in hardware, but traps the processor every time a cache request cannot complete in the local memory. In these cases, a software protocol handler running on the local processor takes over and executes the cache request. In the SCC-NUMA architecture, each memory block is anchored to a home node and cannot be replicated. In SC-COMA, each memory is managed as a set-associative cache; memory lines are associated with a home node but may replicate freely, as in a multicache system. SS-COMA is similar to SC-COMA except that memory is allocated in units of pages during replication so that the mapping of lines into memory is done automatically by the MMU. A replicated page-frame is allocated empty and valid memory lines are brought in from remote on demand. Thus the memory of SS-COMA is managed as a sector-mapped cache with a sector equal to a page.

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

In the following we first overview the hardware and software support needed by the three architectures in Sections 2 and 3. In Section 4, we discuss the performance trade-offs. Then, in Section 5, we describe the simulation set-up and the benchmarks. Finally, in Sections 6 and 7, we present and discuss our results. We conclude in Section 7.

2. HARDWARE SUPPORT

The basic hardware substrate is a message-passing multiprocessor. Each node is made up of a commodity processor and its associated caches. The caches connect to main memory through a multiprocessor local bus. The network interface with one incoming and one outgoing memory-mapped message buffers resides on the bus. The processor may send messages by writing into the outbound buffer and the network interface generates an interrupt to the processor as soon as it receives a message. In essence, this is the standard network of workstations environment, but this communication support is at least present in any parallel computer. Additionally, a custom hardware, the Access Checking Device (ACD), extends the functionality of the memory controller to implement the fine-grained shared memory. The ACD is described in 2.1.

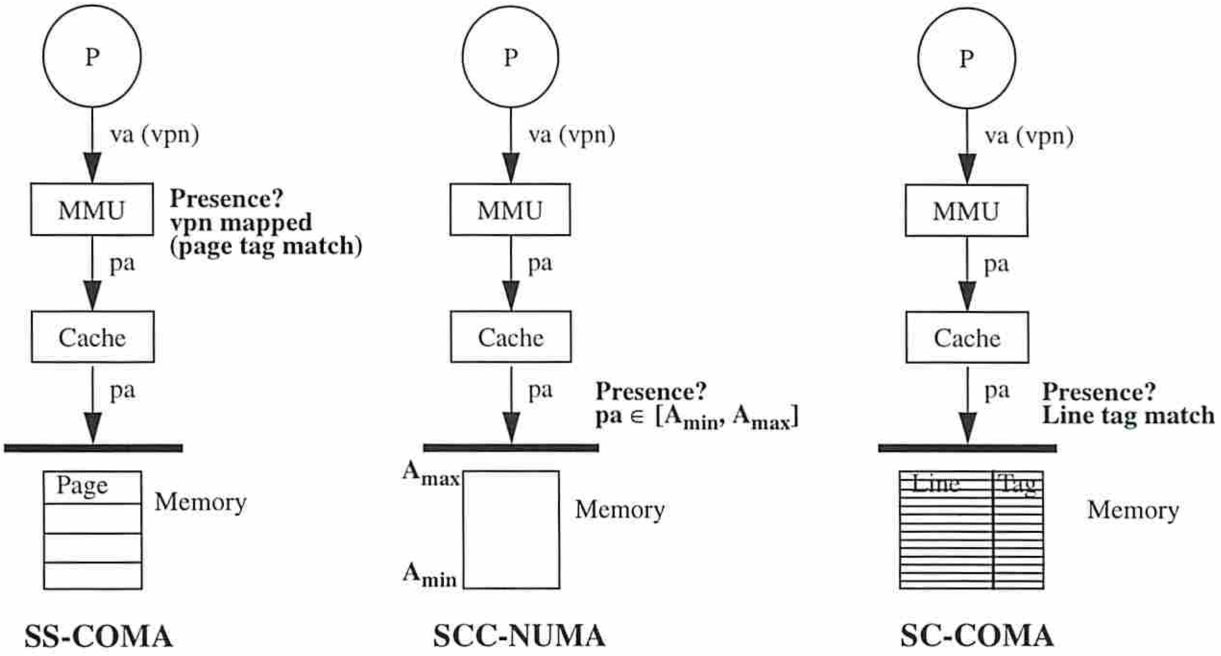
The software protocol handlers execute on the main processor. Hence, we do not rely on the presence of a dedicated protocol processor in the node. We assume the processor features restartable load/store instructions. This is commonly available in modern processors in conjunction with MMU operations. However, given the implementation of the access checking mechanisms, accesses may still trap after passing the MMU check. In this study we assume blocking loads and stores, so that memory access faults are synchronous. Write buffers and asynchronous write faults can be dealt with, provided the handlers have the mechanisms to recover and complete a store from the write buffer.

The cache must support read-only and read-write states, so that a write to a read-only block in the cache triggers a signal reaching the memory. This feature is common in today's multiprocessor-ready processors with on chip caches. Additionally, it must be possible to downgrade the status of a cache line by the software. Either special instructions are available for this purpose (such as *Invalidate* and *Flush*) or appropriate bus transactions are issued by the ACD under software control.

2.1. Memory Access Checking

In a hybrid DSM, every cache request to the shared memory must be validated for completion without software intervention. This process, known as *access checking* or *lookup* [22], employs two mechanisms. The *presence detection* mechanism ascertains the presence of the line in local memory, and, if a local copy is found, it also provides a pointer to the state of the local copy. The *permission checking* mechanism then checks that the state of the line (e.g., Invalid, Shared or Exclusive) is compatible with the request type (e.g., read or write). In order to reduce false sharing effects, at least the resolution of the permission checking mechanism must be fine-grain. Accesses that fail either the presence or permission tests require software intervention.

Figure 1. Implementation of the presence test in each architecture

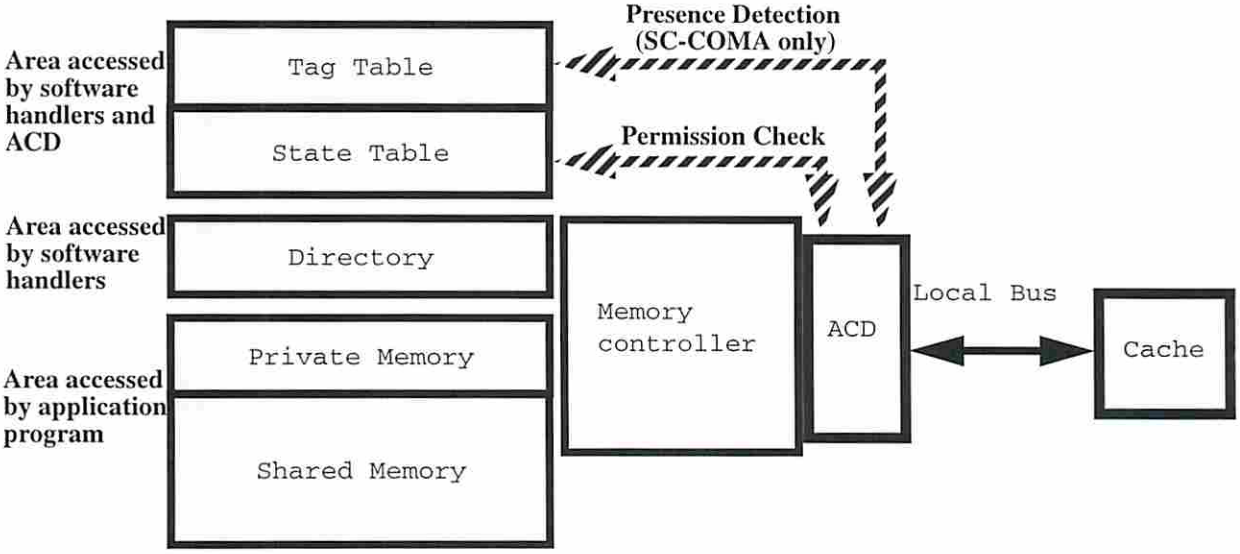


In the three architectures explored in this paper, the permission checking mechanism is identical, and based on a table of state bits indexed by the physical address of the memory line. However, the presence detection mechanisms are very different, as illustrated in Figure 1. SS-COMA uses the standard virtual memory support to implement the presence checking mechanism, at no hardware cost or performance overhead. If a virtual page has an allocated page frame, the datum is present locally at the translated address and the permission check is performed at the memory. Otherwise, a page fault signals the failure of the presence test. Thus, SS-COMA resolves the presence test at page granularity and on every access. Because the sharing space is virtual and physical addresses have only local significance, the coherence protocol must use global identifiers. In the event of an access fault detected by the permission checking mechanism, the physical address must be converted to a global identifier using a reverse translation table.

In the other two architectures the presence test is performed with the physical address on cache misses only. The sharing space is physical and the coherence protocol uses physical identifiers. In SCC-NUMA a partition of the shared space is assigned statically to each processor. A simple comparison of the physical address with the range of local addresses answers the memory presence test. In SC-COMA the main memory has a set-associative organization and a set of tags identifies the lines in the set.

Figure 2 shows the partitioning of the main storage in each node. The Tag and State Tables are consulted by the ACD on every access to check local presence and permission and are modified by the software handlers. The Directory contains the higher-level protocol information and is accessed by the software handlers only. The application memory is divided into private and shared memory. Whereas the State Table and Directory are present in all three machines, the Tag Table is only needed by SC-COMA. In this case, the partition of local memory hosting shared data (often called *attraction memory*) is managed as a set-associative memory. It is simpler --although not necessary-- to combine the Tag and State Tables. Each entry of this combined table packs the tags and the state bits for one entire set. Access checking is performed by fetching one table entry, using the least significant bits of the line address as the set number, and by comparing the tags in the set with the tag of the processor address as well as checking the state bits at the same time. For SCC-NUMA and SS-COMA, we assume the State Table is hosted in a separate memory module, accessed in parallel with the main memory, whereas SC-COMA's Tag and State table is stored in main memory and is accessed prior to fetching a block. In all cases, the Directory is stored in main memory.

Figure 2. Logical diagram of memory modules and the generic Access Control Device (ACD)



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

1. (SCC-NUMA and SC-COMA only) decide if an access must be checked (shared) or not (private), based on the address. If the access is private, skip to 5.
2. (SCC-NUMA only) decide if the bus address points to data present in memory.
3. locate the line and its table entry.
4. fetch the state of the line and validate the access.
5. perform the memory access (in parallel with 4, for SCC-NUMA and SS-COMA)
6. respond with data to the bus transaction OR fault the access using a bus error signal.

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

- memory-mapped registers containing information about the faulted access (address, type, etc.) and control registers.
- assistance for cache block downgrading by issuing bus transactions under software control (optional).

To perform the above functions, the ACD needs a relatively simple finite state machine, several registers, an adder for address computations, and a comparator to match tags and state.

2.2. Special Hardware Support in SCC-NUMA

By contrast with the two COMAs, inclusion is not maintained between cache and local memory in SCC-NUMA. One important aspect of inclusion is that the write-back of victimized lines and the cache miss occurring at the restart of an instruction after an access fault are always completed locally in hardware. Three problems must be solved. First, cache write-backs need special handling when the victim line is non-local. It is unclear whether software recovery of a faulted write-back is feasible, because of the possibility of generating more write-backs during the handling. In any case, it would complicate the handlers, by forcing them to deal with recurrence. Rather, special hardware captures remote cache write-backs and transforms them into network packets sent to the home node. Second, since non-local lines are stored only in the cache, the access fault handlers must be able to inject an incoming line from the network buffers into the cache. Instead of manipulating the cache and its tag storage, we have solved the problem with a special one-line tagged buffer in the ACD. The line is served to the cache when the access is restarted, as if it were coming from local memory and the tag is cleared in the ACD buffer. Finally, special care must be taken with write on read-only copies of remote lines, as the line may be displaced from the cache while the protocol handlers are executing. To solve this, we systematically request a new copy when we upgrade a remote line from read-only to read-write in the cache.

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

2.3. Comparative Hardware Complexity

Of all three solutions, SS-COMA requires the simplest hardware additions. It is limited to the state table and some control in an ACD. SCC-NUMA comes second in terms of hardware complexity. It requires a state table and a slightly more complex control in the ACD, plus some support for remote write backs and cache loads from remote. SC-COMA has the most complex hardware. The state table must be extended to contain tags and some matching logic for the tags must be included in the ACD. However, overall, these mechanisms are very simple, when compared with a full-fledged hardware coherence controller. Last but not least, the two COMAs consume more memory than the CC-NUMA in order to replicate the same line across nodes.

[1] Of course we could assume prefetching in the caches but this would be the topic of a different paper.

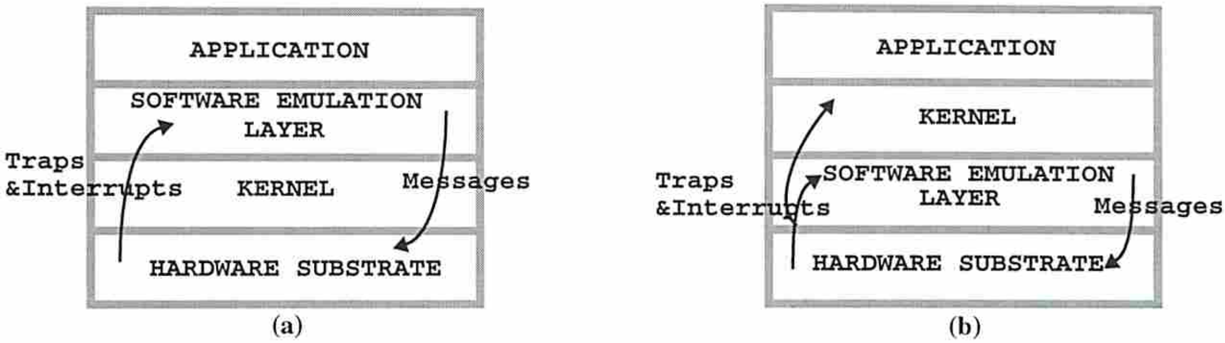
3. SOFTWARE SUPPORT

3.1. Software Protocol Handlers

Implementing software shared-memory raises the problem of choosing the level at which the coherence protocol should be integrated. Traditionally, software and hybrid DSMs have been implemented at the user-level with the operating system providing basic services, such as memory allocation, servicing of messages, and handling of events generated by the hardware level. A miss in the local memory generates a memory exception fault handled by the operating system, which finally generates a signal at the application layer. The handlers of SS-COMA must be implemented at the user-level since they rely on the virtual memory system.

Figure 3. User-level versus kernel-level emulation of shared-memory

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



By contrast, in SCC-NUMA and SC-COMA, which rely on physical addresses only, we implement the software handlers at the kernel level. A kernel-level implementation of a coherence protocol allows the operating system to directly deal with the event without involving the user application layer. Figure 3 illustrates the difference between a kernel-level DSM on one hand and a typical user-level software DSM on the other. An obvious advantage of kernel-level handlers is that the time to handle the events related to shared-memory management is reduced. For example, in the context of network protocols, experience with the x-kernel at the University of Arizona [17] has shown that a kernel-level protocol can outperform user-level protocols by a factor of 2 to 3. Thus, to compete with a kernel-level implementation, a user-level implementation must rely on very careful recoding of parts of the kernel; but even in this case message latencies are still larger [22]. A second advantage is that the protocols are shared by all the processes running on the machine without any overhead. The third advantage is that physical memory management and allocation are not constrained by the virtual memory system.

True, in a user-level protocol, each user could specify a different protocol optimized for each application [22]. However, this feature may be hard to exploit by ordinary programmers, which would rather experiment with ready-made protocols from a library. Similarly, in a kernel-level approach, various (possibly parameterized) protocols are coded by kernel programmers and can be selected by individual users, achieving the same goal without exposing casual users to the intricacies of protocol design and coding.

The software handlers have been written and run on our simulator. 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, in effect supporting processor multiplexing between application and protocol. (This would not be needed if the protocol handlers ran on a dedicated processor.) The prologue also calls the C handler for the corresponding trap. This code is highly optimized, adding an overhead of just 25-30 instructions to the C routines by exploiting SPARC's windowed registers. The context of the application at the time of the exception is saved entirely in processor registers (trap window locals) most of the time. The total sizes of the handlers for the three architectures are shown in Table 1. Clearly these handlers are very small and could easily hold in the first level cache of any modern processor.

Table 1. Total size of protocol handlers (bytes)

Architectures	Access fault handlers	Network interrupt handlers
SCC-NUMA	320	3,248
SS-COMA	360	3,136
SC-COMA	1,640	4,192

3.1.1. Memory exception handler

The memory exception handler has 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. Secondly, the processor enters a spinning loop waiting for a reply to its request. Other requests can interrupt and be handled during this wait. The third part of the handler starts as soon as the processor has received the awaited reply and can start transferring the line from the network buffers.

After the reply reaches the node, the pending memory access should be restarted and committed. However, between the time the reply interruption handler completes and the time the instruction is restarted, another interruption for the same memory line can be raised the line may be removed; the restarting access then triggers a new memory exception. The time window in which a memory line may be invalidated before the processor restarts is called the *window of vulnerability* [13]. To avoid livelocks and ensure forward progress, this window of vulnerability must be closed. We have implemented a software solution. The faulted instruction is re-executed in the memory exception handler, before enabling interruptions and returning to user mode. The instruction is first copied into the code of the handler in a dedicated slot. Then, the corresponding register window for the application is restored and the instruction executed. This is facilitated by the lightweight nature of our handlers. After the epilogue, execution resumes at the instruction following the faulted instruction.

3.1.2. 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; the handler then unlocks the waiting loop in the memory exception handler (polling for the reply is another possibility). No other interruption is allowed within an interruption handler.

3.2. Coherence protocols

The coherence protocol for SCC-NUMA is a classic write-invalidate protocol [8]. Invalidations are issued by and acknowledged to the home node. Transactions involve two or four hops. SS-COMA uses the same protocol with only minor modifications.

The protocol of SC-COMA is a write-invalidate protocol very much like the one in COMA-F [11].

Each memory line can be in one of four stable states: *Exclusive, Master-Shared, Shared* and *Invalid*. In the *Exclusive* (read-write) and *Master-Shared* (read-only) states an *owner identification pointer* located at the *home* node for the line identifies the owner of the memory line. Lines are statically assigned to nodes. The owner responds to read and write miss requests for the line and ensures that at least one copy of the line exists in the system. 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.

A notable difference with COMA-F is that the line owner also maintains the current *copyset* (presence bits) for the line. Thus the owner can send the invalidations without consulting the home node. The pointer maintained by the home node always points to the last node to request ownership. Instead of *Nacking* incoming miss requests to the same line, the future owner keeps these requests in a software buffer. Once the transaction is completed, all the buffered requests are answered. This strategy limits the number of messages exchanged by the nodes as well as the number of interruptions.

Another important difference is that invalidation acknowledgments are collected in the requester node while it is idle waiting for ownership. This optimization saves cycles since no other node is interrupted to collect invalidation acknowledgments.

On a memory replacement, the protocol chooses a victim line according to the following priorities: (1) Invalid, (2) Shared, (3) Exclusive or Master-Shared. To replace an Exclusive or a Master-Shared line, a replacement request carrying a copy of the line is 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 a free frame. 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.

3.3. Support for Paging in SS-COMA

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

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

4. PERFORMANCE TRADE-OFFS

Local transactions in the three architectures are performed at hardware speed. In both SCC-NUMA and SS-COMA, access checking can be done in parallel with the memory access. SC-COMA's access checking method incurs the overhead of tag comparison. This operation must precede the actual data access because the precise position of the line in a set is unknown. Even when the tag table is stored in a separate memory, the actual line transfer cannot proceed in parallel, unless main memory is wide enough to provide the whole set in one access. This is clearly an option when the attraction memory is direct-mapped.

Overall, the performance of the three DSMs is largely dominated by the latency of remote transactions, which require software intervention. We now discuss several aspects of these software transactions.

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

Memory Pressure. A marked advantage of SCC-NUMA is that shared memory is only needed to store the shared data set. In a COMA, memory must also be provided for replication. The ratio between the size of the application's shared data space and the total memory allocated is called the *memory pressure*. When memory pressure increases, replication is more constrained, which in turn causes ping-ponging of memory lines between nodes. Memory pressure increases the node miss rate and the rate of remote node write-backs. In SS-COMA the replication is less efficient than in SC-COMA because of memory fragmentation: Some of the lines reserved when a page is replicated are never accessed or are accessed very little but the entire page frame is nonetheless committed. This memory fragmentation introduces *memory pressure inflation* and increases the page fault rate.

Node Hit Rate. In a hybrid DSM, remote transactions are very costly, thus the overall node hit rate (fraction of processor accesses which remain local) is critical. In SCC-NUMA, there is no caching in the memory. Thus the node hit rate is highly dependent on the cache hit rate and on the placement of the shared pages in memory. By contrast, the node hit rate is not affected by the hit rate of the cache in a COMA (because of inclusion). However, since coherence and cold misses in the node are independent of the amount of caching in the node, only the replacement misses (both capacity and conflict misses) that are remote in SCC-NUMA can be cut in a COMA [24]. The node hit rate is affected by memory pressure,

Node Replacements and Write-backs. In the COMAs, cache write-backs are always local and done in hardware, whereas in the CC-NUMA, they can be remote thus creating interruption overhead at the home. Memory write-backs are executed in software in both COMAs. In SC-COMA, a line is written back after the request for the new line has been sent out. Thus, replacements (albeit complex) are dealt with while the processor is idle, waiting for the new line. The write-back is not in the critical path of the processor, and only creates overhead in the processors that are interrupted by the write-back, as in SCC-NUMA. This is not true in SS-COMA. Memory write-backs occur whenever a replicated page is deallocated. *Before* a page frame can be deallocated, all its dirty lines must first be written back by the fault handler and write-backs occur in bursts. Deallocation in page chunks also creates the false replacement effect [20], i.e. memory lines are written back while they are still actively accessed.

Data Placement Effects. In all three architectures, every page in the shared address space has an associated home node, for which it is a *local* page. In the absence of replication, data from local pages can be accessed much faster than data on remote pages. Even when page replication is supported, a careful placement of pages can improve the performance in two ways. First, memory consumption by replication is reduced when pages are placed in nodes that are guaranteed to access them often. This leaves more room to replicate other pages. Second, the protocol is more efficient by cutting the number of request forwarding that a home node must do. In a hybrid DSM, this also reduces the protocol processing overhead.

Page Faults. This overhead is only incurred in SS-COMA. Unfortunately, it is on the critical path of the processor.

Effect of Memory Line Size. The memory line size for the two COMAs may be selected independently from the cache line size. A larger memory line size may have positive or negative effects. In some cases the prefetching effect in memory cuts down the number of node misses. In other cases, the cache miss rate may increase because of the extra cache invalidations to maintain inclusion when a memory line is replaced.

In order to quantify and compare these effects we have run simulations of the SPLASH-2 benchmarks running on the three architectures. In the following, we describe the experimental setup and then discuss the simulation results for six of the benchmarks.

5. EXPERIMENTAL FRAMEWORK

Our performance evaluation of several architectures is based on execution-driven simulation. In this evaluation, we employ different data placement strategies.

5.1. 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 (crossbar), 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. Statistics are collected about all the relevant events and execution time components.

The processor simulator is a SPARC interpreter. It features trapping on exceptions raised by the simulated memory through the MEXC line. Traps are generated for over-flows and under-flows of the eight register windows as well. Also supported are leveled asynchronous processor interrupts, used for interfacing to the interconnection network. With every invocation, the simulated processor advances exactly by one 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. The binaries of both the software coherence handlers and the application code are executed completely and faithfully by the simulator.

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

5.2. Baseline Architecture

The simulated architecture consists 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, one for instructions (IC) and one for 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 [25], while at the same time yielding a reasonable ratio of memory versus cache sizes for the COMA runs. (The shared memory size per processor in these runs varies between 100 Kbytes to 2Mbytes).

The SLC is connected to the memory by a 128-bits local bus running at 50 Mhz. The memory on each board is 4-way interleaved with a 16-byte wide interface and an access time of 28 cycles (140 ns). The critical word of the line is fetched first. The memory line size is 128 bytes for the COMAs and 64 bytes for the CC-NUMA. Access checking to the code and private data segments is disabled. For SCC-NUMA and SS-COMA, shared data access checking proceeds in parallel with the memory access, adding no overhead. For SC-COMA, the memory controller implements a 4-way set associative attraction memory. (Another study has indeed shown that an 8-way set associative memory does not perform better [1]). The tags and states of the lines in each set are packed in a single double word (8 bytes), which is fetched and checked in 28 cycles (one full memory cycle time) prior to the actual memory access. In all cases, the access to the state (and tag) table by the protocol handlers is uncached and takes 40 cycles. Read latencies are given in Table 2. The memory page size is 4Kbytes in all cases.

The network interface is controlled through a set of memory-mapped registers. A 128 byte line is transferred between the 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. The memory line size in SCC-NUMA is 64 bytes and latencies are adjusted as appropriate. Ten cycles are added for processing at the reception. For ATM networks, the latencies would be considerably longer and therefore the penalties introduced by the software protocol would be relatively much smaller than reported here.

Table 2. 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

5.3. 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 3.

Table 3. Benchmark characteristics

Code	Problem size	Shared space (4 KB pages)
Barnes	8K particles	631
FFT	64K points	868
LU	512x512 matrix, 16x16 blocks	516
Ocean	258x258 ocean	3998
Radix	1M integers, radix 1024	2454
Raytrace	teapot	832

5.4. Data Placement Strategies and Memory Pressure Control

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

For architectures like SS-COMA and SC-COMA, in order to allow for data replication, the amount of allocated memory across all processors must exceed the total requirements of the application. Every node in SS-COMA has a pool of page frames providing the extra space to replicate remote pages. In SC-COMA, every set in the set-associative memory has unoccupied frames. For uniformity sake, we measure the application's shared space size in pages, although a finer-grain could be used for SC-COMA.

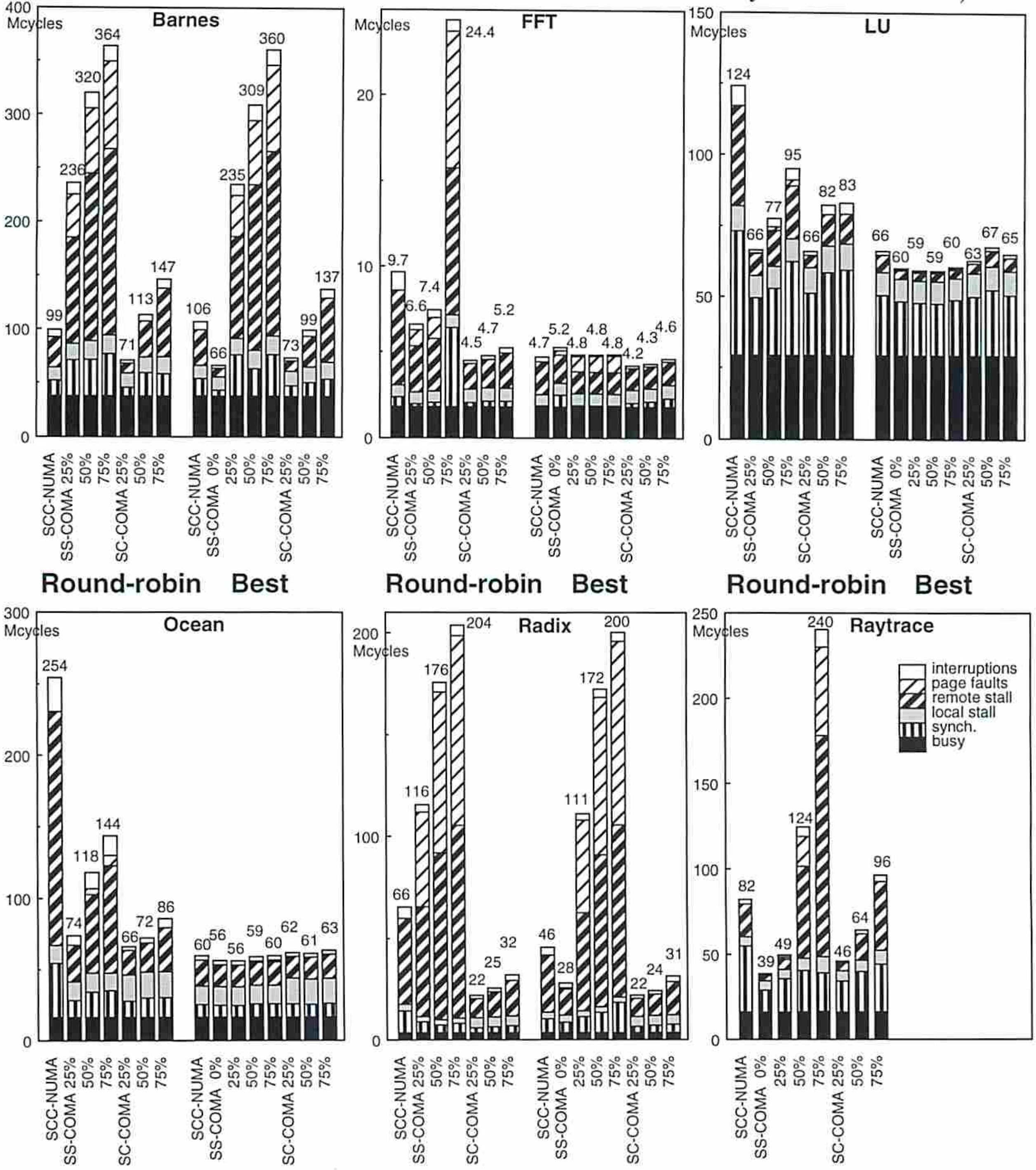
Memory pressure is controlled by setting the number of page frames for each node in SS-COMA and the number of sets in the associative memory in SC-COMA. For SC-COMA, the mapping of a page onto the physical address space not only determines the home node for that page, but also the group of sets where lines can reside in the set-associative memory. In the absence of full associativity, the issue of memory pressure must be dealt with at the level of each set. Consequently, the allocation strategy uses skewing to achieve a uniform pressure across all sets.

6. SIMULATION RESULTS

We now compare the relative performance of the three hybrid architectures. Figure 4 shows the execution times and their components. For SS-COMA and SC-COMA, we include measurements taken at several memory pressure points. The 0% memory pressure for SS-COMA is an extreme case which assumes infinite memory. On the left side of each plot, results correspond to the round-robin placement; on the right, to the best static placement.

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

Figure 4. Execution times for three hybrid architectures (64KB 4-way set-associative cache)



The characteristics of each architecture directly or indirectly affect each of the execution time components, except for the busy time. The direct effects come mostly from the fact that a cache miss either hits in the local memory, adding a small contribution to the local stall, or misses, adding a large contribution to the remote stall. Additionally, an access may incur delays due to a page fault in SS-COMA. Essentially, the direct effects are a matter of event counts and associated time penalties. Table 4 lists the fraction of shared accesses that generate an access fault assuming a best placement. This measure is indicative of the direct contribution of accesses to the local and remote stalls.

The indirect effects are due to contention created by intense protocol overhead, write-back/replacement activity, and modified cache hit ratios. Contention increases queuing delays of protocol requests, thereby the remote access latency. It affects the load balance as well, hence the synchronization delays. The write-back/replacement activity creates an overhead at the reception and, in SS-COMA, at the sender as well. Table 5 gives the counts for these events. Finally, when the write-back/replacement unit is larger than a cache block, the cache hit ratio can be affected negatively (because of inclusion). Another minor effect on the cache hit ratio is due to the protocol handlers' interference.

To simplify the discussion, we compare the results for SCC-NUMA and for SS-COMA to the results for SC-COMA for each benchmark. When comparing SCC-NUMA to SC-COMA, we pay particular attention to the following factors: fraction of replacement misses in the application, existence of a good page placement for the application, memory hit latency, and prefetch/inclusion effects. To indicate the full potential of SS-COMA, Figure 4 also contains results assuming infinite memory (0% memory pressure), where page replacements disappear. Table 7 indicates the actual amount of memory, relative to SCC-NUMA, required to achieve this condition.

In **Barnes**, we see very little effect of the page placement strategy, because of the dynamic nature of work scheduling among the processors. According to the SPLASH-2 report [25], Barnes contains only a moderate amount of capacity misses and it is expected to perform relatively well on a CC-NUMA.

Barnes' main shared data, bodies and cells, are fine-grained and spatial locality is not good. As the simulated galaxy evolves from one time step to another, the assignment of bodies to processors and the configurations of the cells change, leading to dynamic access patterns. The fine-grained attraction memory in SC-COMA can satisfy many of the cold or capacity cache misses, with limited replacements. By contrast, SS-COMA performs poorly even under a memory pressure of 25% (i.e., four times as much memory as the SCC-NUMA). At this pressure, the main culprit is the page faulting overhead and its effect on synchronization because of the load imbalance. Additionally, at higher pressure, the pressure inflation swells the amount of remote stalls as well. Tables 4 and 5 indicate clearly that replication is so inefficient that, even with four times as much memory, the remote traffic is higher than in SCC-NUMA. The explanation is simple: Table 6 shows only three line misses per page fault, indicating that the memory is grossly underutilized (there are 32 lines on a page). With a page granularity for allocation, Barnes really needs a lot of memory in order to gain from data replication. SC-COMA is clearly the best when memory pressure is less than 50%. This shows the effectiveness of fine-grain associativity even after tag checking is accounted for.

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

SS-COMA is very sensitive to the page placement, just like SCC-NUMA. Under best placement, SC-COMA shows slightly higher local stall than SS-COMA, due to tag checking. However this performance effect is dwarfed by the page faulting overhead, caused by cold misses even at low memory pressures. As shown in Table 4, the node hit rates are equal for the two COMAs; it is just that each miss in SS-COMA is much more costly because many of them involve a page fault (see Table 6). Interestingly, SS-COMA performs worse in infinite memory. The reason is that processor 0 is able to accumulate a lot of remote data during the sequential phase of initialization, which creates a bottleneck when other processors request ownership. With limited memory, many page replacements work like self-invalidations. Unlike Barnes, FFT is optimized to reduce false sharing at the page level and temporal locality of accesses to remote pages is very good. This makes SS-COMA almost immune to memory pressure, although the page replication factor is very high.

Table 4. Node miss ratio for shared data (%) (Best placement)

	SCC-NUMA	SS-COMA			SC-COMA		
		25%	50%	75%	25%	50%	75%
Barnes	2.65	6.12	9.59	10.38	0.60	1.69	3.27
FFT	0.68	0.34	0.34	0.34	0.34	0.34	0.34
LU	0.086	0.036	0.036	0.038	0.039	0.064	0.056
Ocean	0.40	0.31	0.32	0.32	0.29	0.29	0.31
Radix	3.98	5.75	9.23	10.52	1.34	1.45	1.78
Raytrace (RR)	1.23	0.33	2.67	5.91	0.24	0.74	1.76

Table 5. Write-back/replacement counts (10^3) (Best placement)

	SCC-NUMA	SS-COMA			SC-COMA		
		25%	50%	75%	25%	50%	75%
Barnes	55.5	72.1	90.1	109.3	0.5	3.4	13.2
FFT	0	0	0	0	0.2	0.3	0.6
LU	0	0	0	0	0	0.1	0.6
Ocean	7.7	0	1.8	2.1	0	0.1	3.5
Radix	657.9	979.8	1659	1908	2.6	31.3	192.9
Raytrace (RR)	6.4	3.2	15.9	29.2	0.6	1.0	15.6

Table 6. Average number of block faults between page faults in SS-COMA (Best placement)

	SS-COMA		
	25%	50%	75%
Barnes	3.02	3.19	2.63
FFT	1.67	1.67	1.67
LU	23.17	21.61	20.0
Ocean	32	22.1	17.77
Radix	1.07	1.02	1.01
Raytrace (RR)	6.6	3.97	3.03

Table 7. Page replication factor in SS-COMA
(ratio between the number of allocated page frames in SS-COMA with infinite memory and SCC-NUMA)

Barnes	FFT	LU	Ocean	Radix	Raytrace
12.3	19.4	7.5	2.7	18.7	11.2

LU has a high fraction of cold/coherence misses, which would make it an ideal candidate for SCC-NUMA. But, just like FFT, the cache miss ratio doubles when the line size is 64 bytes, as compared to 128 bytes, resulting in more remote accesses and increased synchronization delays.

The node miss and write-back rates in LU are so tiny that effects of the memory systems of the two hybrid COMAs are marginal. The dominant part of the execution time is made of busy time plus synchronization time. Both architectures benefit slightly from careful page placement. Tag checking and lower efficiency of the coherence protocol explain the increased local and remote stall in SC-COMA. Page-faulting activity is insignificant and SS-COMA is the overall winner.

It is well known that **Ocean** does very well on a CC-NUMA under best placement. The scheduling is static and the spatial locality is large. Ocean has a large number of capacity cache misses but the misses are mostly local under best page placement. This explains its good performance on SCC-NUMA. The performance of SC-COMA suffers slightly from the tag checking overhead on memory hits.

SS-COMA is able to satisfy the tiny fraction of capacity misses to remote data with little page faulting overhead. This is due to the coarse grain of the data structures, exceeding page size. Memory utilization is excellent. SC-COMA brings a small improvement in the node hit ratio, but pays a higher price for the bulk of local memory hits. Overall, the fine-grain associativity of SC-COMA is not needed in this application. However, note again that SC-COMA is the architecture least sensitive to page placement.

Radix has significant capacity misses with a 64 KB cache. Because the memory access pattern of each process is data-dependent, the best placement cannot improve the execution time to a point that SCC-NUMA is competitive with SC-COMA. The poor spatial locality also creates a large number of remote cache write-backs, increasing the fraction of time spent in interruptions and synchronizations.

Radix is a disaster for SS-COMA. The reason is the permutation phase where, even with the best placement, an overwhelming fraction of writes are to non-local data. Because accesses are scattered in this phase and the data structures to be updated are fine-grained, a large number of pages have to be mapped and subsequently deallocated to make room for others. In effect, a single line is used throughout the lifetime of a such page (see Table 6). The performance of SS-COMA can only be improved at the cost of much higher memory consumption: for the 0% pressure point, almost 19 times more memory is necessary than in SCC-NUMA.

Raytrace can benefit mostly from the replication of the scene, the main data structure, which is read-only. The scheduling of the work is somewhat dynamic and the accesses through the scene data are data-dependent. Thus no best placement was recommended by the programmer. There are significant capacity misses, which favors SC-COMA. Again, a coherence unit size of 128 bytes in SC-COMA adds an additional prefetching benefit.

Objects are fine grain and there is little spatial locality, so that replication is more effective in SC-COMA than in SS-COMA. This effect is visible at higher memory pressure.

7. SUMMARY

SCC-NUMA's advantages are relative hardware simplicity, parsimonious utilization of memory, and fast local memory hit access. Whether SCC-NUMA is better depends mostly on the number of remote replacement misses in the cache and the optimal size for the coherence unit. SCC-NUMA's performance is very sensitive to the page placement in each node. The page placement cannot be improved significantly when the scheduling is dynamic or when the data access pattern has very little spatial locality. Overall, SCC-NUMA cannot best SC-COMA at 50% memory pressure, especially if we want the programmer to be oblivious of the idiosyncrasies of the machine. The hardware cost of SC-COMA at 50% pressure is heavy though: twice the memory consumption and a tag memory.

The hardware of SS-COMA is even simpler than SCC-NUMA and the local memory hit access is just as fast. However SS-COMA's consumption of memory is prohibitive for applications with fine-grain sharing because of the coarse allocation of memory at the time of replication and the memory pressure inflation. As a result, under finite memory constraints, the page fault rate and the node miss rate soar as processors access fine grain data residing in remote memories. The page fault overhead is very significant in some cases. (Two previous studies either assumed a very low penalty for a page fault [20] or very low memory pressure [19].) Table 6 shows the average number of memory misses between two page faults. The numbers correspond to the best placement. Except for Ocean and LU, the page size grain clearly leads to fragmentation and it is a direct factor of reduction in the memory hit ratio. The only applications where SS-COMA does slightly better are Ocean and LU. However, these application have coarse grain sharing and would probably perform well on a pure software DSM. Meanwhile, there is no real indication from our results that SS-COMA is effective at dealing with fine-grain sharing in the general case, unless large amounts of memory can be used without concerns for efficiency. In fact, even with 4 times as much memory, it does not seem to be a clear winner against SCC-NUMA with good page placement.

Overall, SC-COMA shows consistent good performance across benchmarks and page placement strategies and is only narrowly beaten by other machines when the application exhibits coarse-grain sharing and high spatial locality, mostly because of the slower memory access time on a hit. The hardware cost is higher than the other two machines because of the tag memory. We note that SC-COMA is much more stingy in memory consumption and less sensitive to memory pressure than SS-COMA. Even under 75% memory pressure SC-COMA does very well.

8. RELATED WORK

To alleviate problems with page-level sharing in software DSMs, some systems abandon the use of the virtual memory support for access checking and perform fine-grain access checking implemented entirely in software: Blizzard-S [22], Shasta [21]. In the absence of adequate compilers, the application executable can be modified by preceding load/store instructions with a short code to perform the checking. While maintaining all the flexibility and portability of a software implementation, such systems incur an execution overhead of up to 35% in Shasta [21], additional to the protocol processing overhead. Furthermore, the consumption of memory can be significant because every page of shared data replicates to every processor using any part of the page for the entire lifetime of the application. Memory-informing operations [10] can help further reduce the access checking overhead in software-only DSMs, but require special processor support, currently unavailable.

Alewife first prototyped a version of hybrid DSM. Software-extended protocols [4] were used to build a CC-NUMA machine where the simple features of the protocol were supported by hardware and the infrequent, but complex, situations handled in software. Grahn and Stenström later proposed the software-only directory protocols [8], a version of SCC-NUMA with extensive hardware assistance from a custom node controller. It employs the same idea of separating the state table, needed by the hardware to do access checking, from the directory data, used by the software handlers to maintain presence information. Remote accesses are handled completely in hardware (interrupts are generated only at the home nodes). This feature requires high-availability processor interrupts and an increased level of complexity in the node controller.

Blizzard [22] is an SS-COMA implementation on the CM-5 which uses a combination of ECC bits in the memory and page access right bits in the MMU to detect block access faults. Because only one bit is available in the ECC (encoding valid/invalid states), write operations usually take an expensive page fault, due to the page being mapped read-only, and resolve the access by checking extra tables for write permission. Thus, the latency of write hits is considerable. Blizzard uses a low-overhead method of switching the processor between application and protocol processing, as in SC-COMA. However, because of insufficient privileges at the user level, it must find ways around several idiosyncrasies of the operating system, with some impact on performance. Blizzard's provision for guaranteeing forward progress is unclear. Faulting instructions are re-executed after returning from the exception, which leaves open a window of vulnerability. Efficient communication is supported by CM-5's user-level network interface, a non-standard feature in the large-volume workstation market.

Typhoon-0 [19] implements another SS-COMA on a SparcStation-20. A custom MBus device, the Vortex, supports access control by using a full-map directory. In contrast, SC-COMA implements access checking in the memory controller and stores the directory in regular DRAM. The protocol handlers for Typhoon-0 is designed to run at user-level, offering it additional flexibility. One of the two processors in the workstation serves as a dedicated processor which runs protocols at the user level and detects coherence events through polling of the status registers of the Vortex. While the small overhead of multiplexing the main processor is eliminated, this could be a waste of valuable resources in a small-scale SMP.

START-NG [5] includes an Address Capture Device (ACD) to support fine-grain sharing in a manner similar to Typhoon-0. A level-3 cache is managed in software, which puts a heavier penalty on all level-2 cache misses to both local and remote lines. The coherence protocol runs in user mode on one of the node's four processors. START-NG's dedicated protocol processor and its interaction with the ACD have strong similarities with the solution in Typhoon-0. A custom tightly-coupled network interface can be accessed by processors through their level-2 cache interface.

The Stanford FLASH [14] 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. MAGIC also contains the protocol processor, along with its own instruction and data caches. Currently, like SC-COMA, FLASH does not allow user-level handlers and runs protocol handlers at system-level. FLASH, however, provides support for low-overhead user-level message-passing. The cache coherence protocol is based on the DASH protocol and the directory structure uses dynamic pointer allocation for better scalability. While the performance of FLASH is high, its cost is high as well. One MAGIC chip is required for every processor in the system, even if its utilization is low. Being a dedicated DSM, FLASH and its techniques are inapplicable to current commodity workstations.

Typhoon [18] is another proposal for a DSM with high levels of integration. It is based on Tempest, a parallel machine interface for user-level shared memory, Simple COMA style. Typhoon's 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. Typhoon also provides hardware support for low-overhead messaging, bulk data transfers and virtual memory management. User-level protocol handling provides a maximum degree of flexibility at the cost of having to use a RTLB. This leads to slightly higher miss latencies than system-level handlers and increased costs and complexity. Also, for SMP clusters, a single protocol processor may constitute a bottleneck.

9. CONCLUSION

In hybrid distributed shared memory systems coherence among nodes is maintained in software but hardware support is added to reduce the size of the coherence unit to a cache block to avoid false sharing and thus supporting fine-grain sharing. Still, current hybrid DSM systems allocate memory at the page level at the time of replication. We have shown in this paper that this allocation strategy may defeat the purpose of supporting fine-grain sharing because of memory pressure inflation. For the applications with fine-grain sharing we have seen that even with four times as much memory as a CC-NUMA the outcome may be poor or even disastrous. Thus, in this paper, we advocate to either avoid replication by implementing a hybrid CC-NUMA (called SCC-NUMA) or even better to replicate memory in units of cache lines -- not pages-- and manage the memory as a set-associative cache thus implementing a hybrid COMA (called SC-COMA).

We have compared three hybrid architectures: SCC-NUMA, SC-COMA and SS-NUMA, the hybrid version of the Simple COMA in which memory is allocated at the page level at replication time. Basically we show that, with some additional hardware support, SC-COMA with a memory associativity of four reaches the best (or close to the best) performance level across various applications with widely different behaviors. In particular the performance level of SC-COMA is independent of the page placement and is very good across various memory pressures. This is in contrast to both SCC-NUMA and SS-COMA. In these machines, performance exhibits large variations across applications, and thus the programmer must be aware of the memory organization in order to squeeze acceptable performance levels. In some cases, we have shown that the applications would have to be re-written or the algorithm would have to be re-designed (case of radix) to perform well on SS-COMA at reasonable memory pressures. Overall SC-COMA realizes the goals of ease of programming, high performance, and consistent behavior across applications, data set sizes and page placements.

As hybrid DSMs are slower than hardware DSMs[1], the hybrid DSM approach must offer other advantages such as ease of development and flexibility. The ACD mechanism in the memory controller is a simple design, which can be standardized across generations of machines. In this context, a small performance slowdown can be acceptable if it allows constructors to design, debug, build, test and manufacture large multiprocessors using the latest processors with a shorter time to market. The ACD mechanism can also be attached to the bus of a commodity workstation and used as a snooping device. In this study, we have used a write-invalidate protocol, and thus we have not explored the additional performance improvements possible with user defined protocols. This may be an avenue for future work. (However, we note that user-defined protocols exacerbate the difficulty of programming the machine.)

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

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

A possibility that remains unexplored in this study is to combine tag-free local memory, like in SCC-NUMA, with associative memory for fine-grain replication of remote data, like in SC-COMA. S3.mp [16] is an example of a hardware DSM based on this idea. This would seem to perfectly suit the needs of all benchmarks. Pages with strong processor affinity would be placed into the tag-free local memory, whereas the other pages would be placed in round-robin manner in the associative memory, so that data can automatically migrate.

[1] elsewhere [2], we show that SC-COMA achieves a slowdown of 11-66% with respect to an aggressive hardware implementation.

10. REFERENCES

[1] Anonymous. Design and Performance of a Software-Controlled COMA.

[2] Anonymous. Hardware vs. Software Implementation of COMA: A Performance Comparison.

[3] 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.

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] Steven K. Reinhardt, Robert W. Pfile, David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. 23rd International Symposium on Computer Architecture (ISCA), May 1996.

[20] 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.

[21] D.J. Scales, K. Gharachorloo and C.A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In Proceedings of the 3rd International Symposium on High-Performance Computer Architecture, 1997.

[22] 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.

[23] P. Stenström. A Survey of Cache Coherence Scheme for Multiprocessors. *IEEE Computer*, Vol. 23, No. 6, June 1990.

[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.