

The Impact of Memory Organization in Hybrid DSM

Adrian Moga

IBM
15450 S.W. Koll Parkway
Beaverton, OR 97006-6063
moga@us.ibm.com

Michel Dubois

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
dubois@paris.usc.edu

Abstract

Hybrid Distributed Shared Memory (DSM) systems are shared-memory multiprocessor architectures with software-implemented coherence protocols and hardware support for fine-grain access control. In this study, we compare the design issues and performance consequences for adopting in hybrid DSM four memory organizations inspired from existing architectures: CC-NUMA, RC-NUMA, S-COMA, and COMA. The performance of these architectures has been analyzed for hardware implementations. However, there are many competing trade-offs requiring careful re-evaluations for hybrid implementations. To do so, on a common platform, we have completely implemented the software protocol handlers for the four architectures. Our evaluations are based on detailed execution-driven simulations of six complete benchmarks with both coarse-grain and fine-grain sharing.

1. INTRODUCTION

Software DSM (Distributed Shared Memory) systems (Ivy [15], Treadmarks [5], [30]) implement shared memory on top of a message-passing hardware substrate using commodity hardware and software-only protocols. Typically, their performance is worse than hardware DSM due to overheads and the large granularity of sharing at the page level.

To improve the performance of software DSM for applications with fine-grain sharing several hybrid hardware/software approaches have been explored. In these approaches, some hardware support is provided to assist the software. Examples include SHRIMP [1], Cashmere [12], Blizzard-E [23], Typhoon [19][20], START-NG [3], and Flash [14].

This study explores four memory organizations for hybrid DSM systems inspired from hardware DSMs: CC-NUMA, RC-NUMA (NUMA with remote data cache) [27], S-COMA (Simple COMA) [21], and COMA [25]. The hybrid architectures rely on a memory Access Control Device (ACD) which satisfies memory hits in hardware, but requires software intervention on the local processor every time a cache miss cannot complete in the local memory. In CC-NUMA each memory block is anchored to a home node and cannot be replicated outside the processor caches. This restriction is alleviated in RC-NUMA by a remote data cache (RDC) organized in main memory. In COMA, each memory is managed as a set-associative cache; memory blocks are associated with a home node, but may migrate freely. S-COMA is similar to COMA except that memory is allocated in units of pages so that the mapping of blocks into memory is done automatically by the MMU. A replicated page frame is allocated empty and valid memory blocks are brought in on demand.

There are many performance trade-offs in these four architectures involving mostly the behavior of the memory hierarchy. To quantify and compare these trade-offs we have completely implemented the coherence protocol handlers for each architecture and we evaluate six SPLASH-2 benchmarks [26], exhibiting a combination of fine-grain and coarse-grain sharing, the same execution-driven simulation platform.

In the following we first overview the hardware and software support needed by the four architectures in Sections 2 and 3. In Section 4, we describe the simulation setup and the benchmarks. In Section 5 we present and discuss our results. After a brief review of related work in Section 6, we conclude in Section 7.

2. HARDWARE SUPPORT

The basic hardware substrate is a message-passing multiprocessor, typical for a network of workstations. A custom hardware device, the Access Checking Device (ACD),

extends the functionality of the memory controller in each node to implement the fine-grained shared memory.

The software protocol handlers execute on the main processor, thus obviating the need for a dedicated protocol processor. We assume load/store instructions are restartable. Given the implementation of the access control mechanisms and for simplicity, we assume blocking loads and stores, so that late (after MMU check) memory traps are synchronous. However, the class of hybrid systems evaluated here can also be implemented with ILP processors as demonstrated in a recent paper [29] showing how to efficiently take late memory traps.

2.1. Memory Access Control

In a hybrid DSM, cache requests to the shared memory must be validated for completion without software intervention. This process, known as *access checking* or *lookup* [23], employs two mechanisms. The *presence detection* mechanism ascertains the presence of the line in local memory, and, if a local copy is found, it also fetches its state. The *permission checking* mechanism then checks that the state is compatible with the request type. In order to reduce false sharing effects, at least the resolution of the permission checking mechanism must be fine-grain. Accesses failing either the presence or permission tests require software intervention.

The permission checking mechanism is generally based on state bits associated with every physical memory line. However, the presence detection mechanisms are very different, as illustrated in Figure 1. S-COMA uses the standard virtual memory support. If a virtual page has an allocated page frame, the datum is present locally at the translated address. Otherwise, a page fault signals the fail-

ure of the presence test. Thus, S-COMA resolves the presence test at page granularity and on every access. Because the sharing space is virtual and physical addresses have only local significance, the coherence protocol must use global identifiers. In the event of an access fault detected by the permission checking mechanism, the physical address must be converted to a global identifier using a reverse translation table.

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

Figure 2 shows the partitioning of the main storage in each node. The Directory is stored in main memory and contains high-level protocol information. The Tag and State Tables are consulted by the ACD to check local presence and/or permission and are modified by the software handlers. Whereas the State Table is present in all four machines, the Tag Table is only needed by COMA and RC-NUMA. In COMA, the partition of local memory hosting shared data, often called attraction memory (AM), is managed as a set-associative memory. It is simpler to combine the Tag and State Tables and to pack the information for an entire set in a single entry. For CC-NUMA and S-COMA, we assume the State Table is hosted in a separate

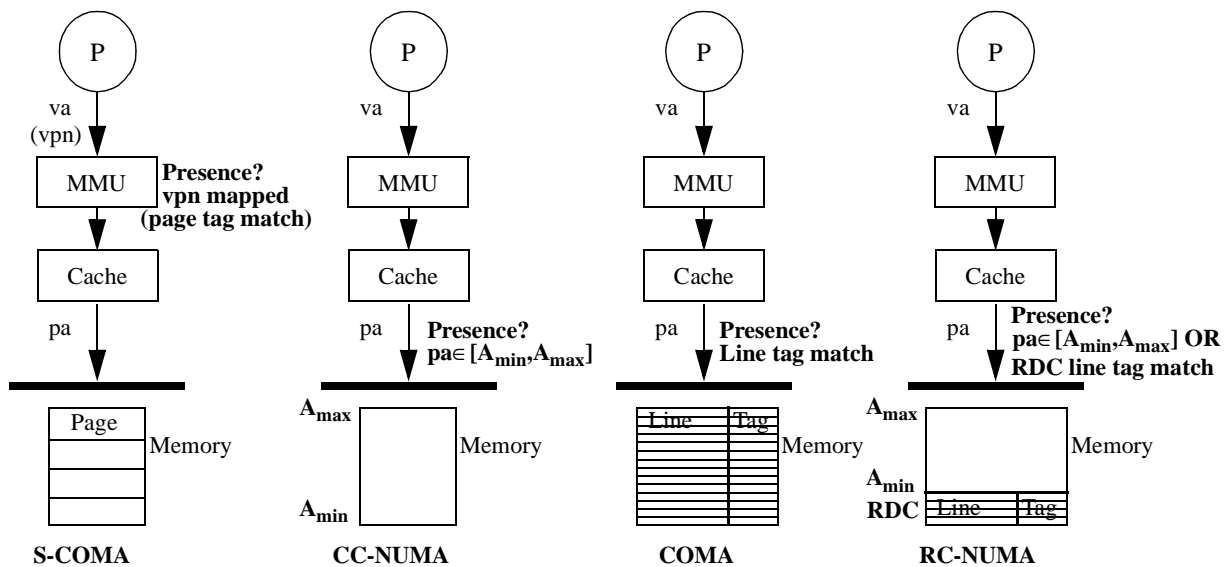


Figure 1. Implementation of the presence test in each architecture

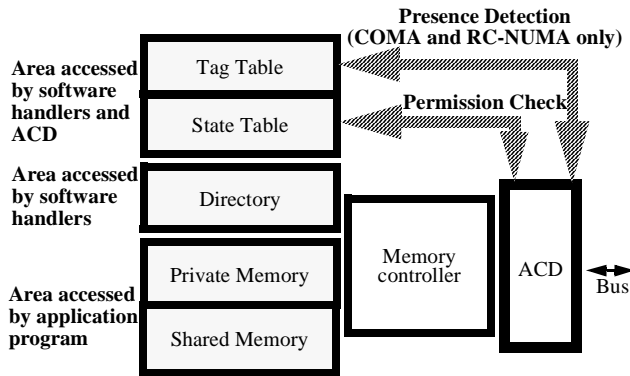


Figure 2. Logical diagram of memory partitions and the generic Access Checking Device (ACD)

memory module, accessed in parallel with the main memory, whereas COMA's Tag and State table is stored in main memory and is accessed prior to fetching a block. RC-NUMA has both a State Table for the local memory, like CC-NUMA, and a Tag and State table, like COMA, for the RDC. Concretely, the ACD is either incorporated in the memory controller or in a snooping device attached to the memory bus. It performs the following services on every memory access:

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

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

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

2.2. Special Hardware Support in CC-NUMA

Because inclusion is not maintained between cache and local memory in CC-NUMA the write-back of victimized lines and the cache miss occurring at the restart of an instruction after an access fault need special hardware support.

First, special hardware in the network interface must

capture the write-back of non-local cache blocks and transforms them into packets sent to the home node. Second, we augment the ACD with a special one-entry buffer to store a missing remote cache block. The block is served to the cache when the access is restarted, as if it were coming from local memory. Finally, special care must be taken when we upgrade a remote block from read-only to read-write in the cache, as the block may be displaced from the cache while the protocol handlers are executing. To solve this, we systematically request a new copy.

Basic and impossible-to-fix hardware shortcomings of CC-NUMA stem from the fact that the local memory cannot host remote lines. First, the coherence unit between nodes must be the cache line, thus disabling the performance benefits of prefetching associated with a larger fetch unit in memory. Secondly, CC-NUMA is vulnerable to cache conflicts which victimize remote blocks. Finally, the total amount of remote caching in the node is limited to the size of the secondary cache.

3. SOFTWARE SUPPORT

3.1. Software Protocol Handlers

Traditionally, software and hybrid DSMs have been implemented at the user-level with the operating system providing basic services, such as memory allocation, messaging, 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. S-COMA's handlers must be implemented at the user-level since they rely on the virtual memory system.

By contrast, in CC-NUMA and COMA, which rely on physical addresses only, we can implement the software handlers at the kernel level without involving the user application layer (as shown in Figure 3). Some advantages of kernel-level handlers are: they take less time to handle coherence events, they are shared by all the processes running on the machine, and physical memory management and allocation are not constrained by the virtual memory system.

Our software handlers are composed of three parts: a prologue, a C handler, and an epilogue. The prologue and epilogue save and restore the processor state, in effect supporting processor multiplexing between application and protocol. The prologue calls the C handler for the corresponding event. This code is highly optimized, adding an overhead of just 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 most of the time. The total size of the handlers is 3-5 KB and they could easily fit in the first

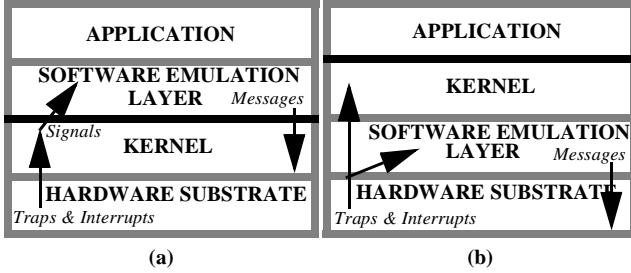


Figure 3. User-level versus kernel-level emulation of shared-memory

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 level cache of modern processors.

Memory exception handler. The memory exception handler has three parts. First, based on information recorded by the ACD about the faulting access, the processor builds a request which is sent to a remote node. Secondly, the processor starts spinning while waiting for a reply. Other requests can interrupt and be handled during this wait. Before spinning, COMA and RC-NUMA perform allocation in the AM/RDC, possibly sending out replacement or write-back requests. The third part begins when the node has received the awaited reply.

After the requested line is stored into memory, the pending memory access should be restarted and committed. To avoid livelocks and ensure forward progress, the window of vulnerability [13] must be closed. Otherwise, an external request might remove the line. We have implemented a software solution. The faulted instruction is re-executed in the memory exception handler, before enabling interruptions and resuming user mode. This is facilitated by the windowed registers and the lightweight handlers.

Interruption handler. Interruptions are triggered by the network interface when external messages (requests or replies) are received. If the message is a request, the handler treats the request and sends a reply. 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. No other interruption is allowed within an interruption handler.

3.2. Coherence protocols

The coherence protocol for CC-NUMA is a classic write-invalidate protocol [8] with two or four hops. Invalidations are issued by and acknowledged to the home node.

The protocol of COMA is a write-invalidate protocol similar to COMA-F [11]. Each memory line has a statically assigned home node and can be in one of four stable states:

Exclusive, Master-Shared, Shared and Invalid. In the Exclusive and Master-Shared states, an owner pointer, indicating the last writer, located at the home node identifies the owner of the memory line. The owner responds to requests forwarded by the home node and ensures that at least one copy of the line exists in the system.

A notable difference with COMA-F is that the owner also maintains the current presence bits for the line. Thus the owner can send invalidations without consulting the home node. Miss requests forwarded to a would-be owner (before the ownership transaction is closed) are kept in a software buffer and replied to just after the ownership transaction completes. This strategy reduces the number of messages exchanged by the nodes as well as the number of interruptions. Another optimization is that invalidation acknowledgments are collected at the requestor node while it idles waiting for ownership. Replacement policies are implemented in RC-NUMA and COMA for the management of the RDC/AM [28].

3.3. Page Cache Management in S-COMA

Some of the page frames in S-COMA’s memory permanently host shared data that is placed locally, while others replicate pages from remote nodes. The management of the page cache is done by a custom device driver, invoked on every page fault. The driver allocates page frames, flushes deallocated pages, initializes the state table, updates the MMU hardware and system tables, and maintains the physical-to-global reverse translation table.

4. EXPERIMENTAL FRAMEWORK

4.1. Simulator and Benchmarks

The processor simulator is a SPARC V7 interpreter. It does not simulate the details of the instruction pipeline. The simulator handles synchronous memory access exceptions and asynchronous interrupts generated by the network interface. For S-COMA, the actions of the page fault handler are performed by the simulator and we suspend the faulted processor for the estimated duration (1100-1500 cycles on average).

Table 1: Benchmark characteristics

Code	Problem size	Shared space (MB)
Barnes	16K particles	3.94
FFT	64K points	3.54
LU	512x512 matrix, 16x16 blocks	2.16
Ocean	258x258 ocean	15.52
Radix	1M integers, radix 1024	9.87
Raytrace	car	34.86

The characteristics of the benchmarks are given in

Table 1. In order to show the importance of data placement, we performed our experiments under two strategies: round-robin (RR) and best placement. For S-COMA, RC-NUMA, and COMA, we also vary the amount of memory allowed for replication.

4.2. Baseline Architecture

The simulated architecture consists of 32 nodes, each with a 200 MHz SPARC processor, a hierarchy of virtually-indexed caches, memory, and a network interface. The 4KB first-level cache is direct-mapped with 32 bytes blocks. The 16 KB second-level cache (SLC) is four-way set-associative and fits the primary working set WS1 of the benchmarks [26].

A memory line is 64 bytes for CC-NUMA and 128 bytes for the others. Tag-free shared data access checking proceeds in parallel with the memory access without overhead. COMA's AM is 4-way set-associative. The tags and states for a set are packed in 8 bytes, which are fetched and checked in 28 cycles (one full memory cycle time) prior to the actual data access. Uncached access to the state (and tag) table by the protocol handlers takes 40 cycles. Read latencies are given in Table 2. RC-NUMA's RDC has the same organization and parameters as COMA's AM. The memory page size is 4 KB in all cases.

The network interface can transfer a 128-byte line to/from memory in 80 cycles. The network is a crossbar with constant latency between any two nodes. The transfer of an 8-byte request takes 16 cycles and the transfer of a 128-byte line takes 272 cycles (latencies are adjusted for CC-NUMA). Ten cycles are added for processing at the reception.

Table 2: Read latencies

Read Requests	Latency (pclocks)
Read served by FLC	1
Read served by SLC	7
Read served by the tag-free local memory	46
Read served by the RDC/AM	46+28=74
Direct read access from memory(4 bytes, uncached)	40

5. SIMULATION RESULTS

We now compare the relative performance of the four hybrid architectures. Figure 4 shows the execution times and their components. For S-COMA, RC-NUMA, and COMA, we include measurements taken at several memory overheads: 100%, 50%, and 33%, corresponding to memory pressures of 50%, 66%, and 75%. S-COMA includes an extreme case assuming infinite memory. (At this point, page replacements disappear, but the memory consumption is prohibitive; see Table 7.) On the left side of

each plot, results correspond to the round-robin placement; on the right, to the best static placement. The results are normalized with respect to COMA 50% pressure and round-robin data placement.

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 software intervention, the delay is counted as local stall. The other accesses contribute to the remote stall. An additional component of the execution time is due to the processing of external requests which interrupt the application thread. We do not include in this category the overhead of requests occurring while the processor spins at a synchronization or a pending remote access. Finally, S-COMA has a significant page faulting activity.

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

The indirect effects are due to contention created by intense protocol overhead and write-back/replacement activity. Contention increases queuing delays of protocol requests, thereby the remote access latency. It affects the load balance as well, hence the synchronization delays. The write-back/replacement activity creates an overhead at the reception and, in S-COMA, at the sender as well. Table 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).

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

In **Barnes**, we see little effect of the data placement strategy because of the dynamic work scheduling among the processors. According to the SPLASH-2 report [26], the majority of cache misses are capacity related and CC-NUMA is not expected to perform well. The fine-grained AM in COMA can satisfy many capacity cache misses,

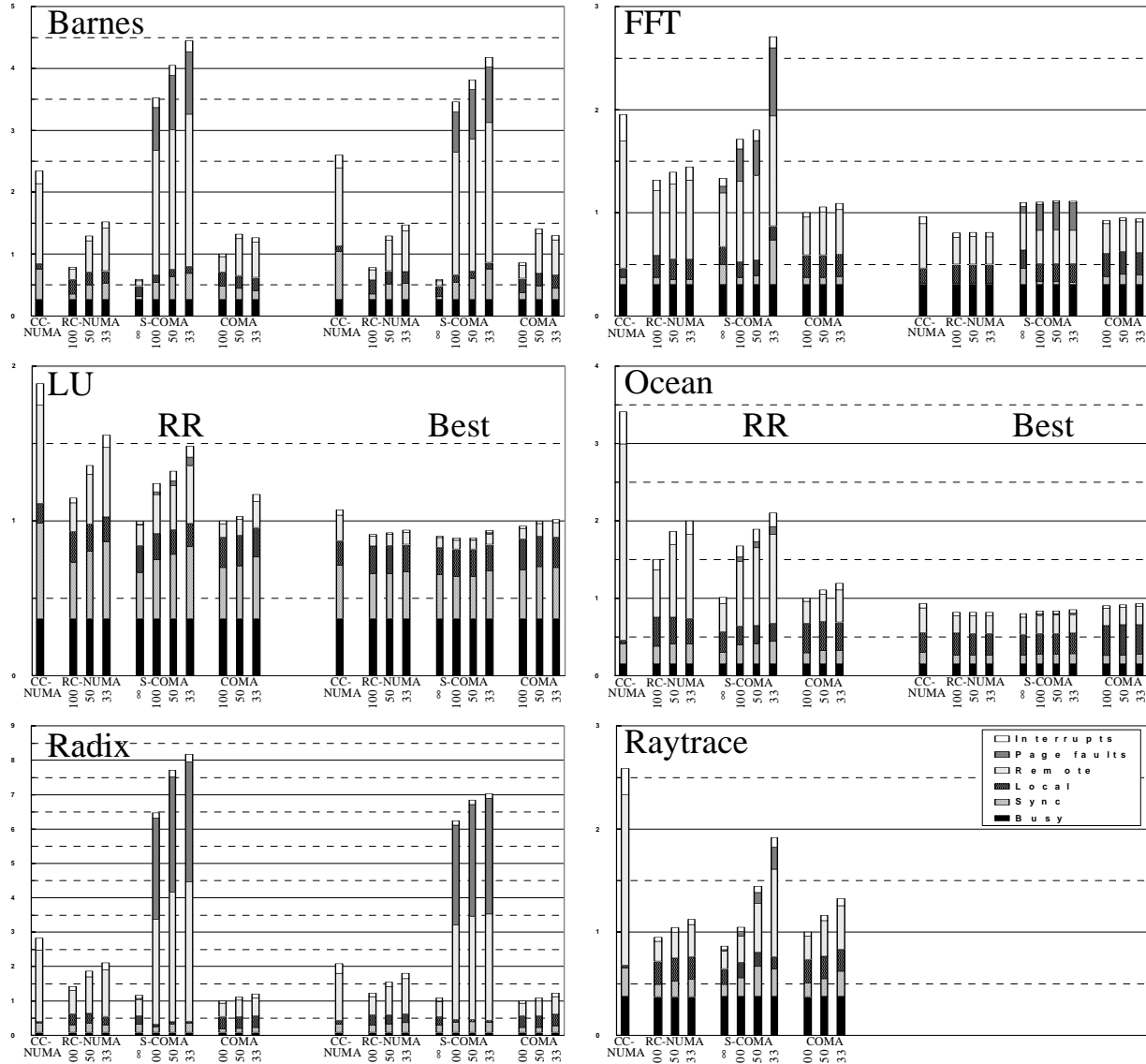


Figure 4. Execution times for the hybrid architectures (16KB 4-way set-associative SLC)

with limited replacements. RC-NUMA's RDC does it just as well at 50% pressure, because it is large enough to accommodate an important working set WS1. As the RDC becomes smaller, WS1 no longer fits and the node miss ratio picks up. Overall, COMA performs the best, especially when memory pressure is high. This shows the effectiveness of the AM even after tag-checking is accounted for.

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

performs poorly even at a memory pressure of 50%. The main culprit is page cache thrashing, which creates a large page faulting overhead and deteriorates the node hit rate. Tables 4 and 5 indicate clearly that replication is so inefficient that, at 75% pressure, S-COMA's node miss and write-back ratios are almost identical to CC-NUMA. The explanation is simple: Table 6 shows only three line misses per page fault, indicating that the memory is grossly underutilized (there are 32 lines on a page). Additionally, there is a negative effect on the synchronization stall, because the protocol's high consumption of processing bandwidth induces load imbalances.

FFT is ideally suited for a CC-NUMA architecture. Its

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

The spatial locality properties of FFT are such that there are absolutely no gains in prefetching blocks larger than 128 bytes, which hurts S-COMA. During communication phases, each processor marches through a large number of remote pages fetching just one 128-byte block from each, thus thrashing S-COMA's page cache. As shown in Table 4, the node hit rates are almost equal for S-COMA, RC-NUMA, and COMA; however, the average miss in S-COMA is much more costly because most of them involve a page fault (see Table 6).

There are two reasons why COMA, under best placement, performs worse than RC-NUMA. First, processor 0 initializes all the data during the sequential phase, which forces some replacements for local data. When the parallel section begins, processor 0 recovers its local data from processor 1. This creates hotspotting on the two processors and increased synchronization penalty for the first lock-step. Secondly, there is the tag-checking overhead for memory hits. Still, COMA outperforms RC-NUMA under the round-robin placement because the AM satisfies more capacity misses.

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

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

It is well known that **Ocean** does very well on a CC-NUMA under best placement. The scheduling is static and the spatial locality is large. Ocean has a large number of capacity cache misses but the misses are mostly local under

the best page placement, thus incurring no software overhead in CC-NUMA. The performance of COMA suffers slightly from the tag-checking overhead on memory hits.

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

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

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

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

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

Fine grain objects and little spatial locality make replication less effective in S-COMA, especially at high memory pressure. Because Raytrace only needs data replication, RC-NUMA has an advantage over COMA, as the RDC can achieve replication with fewer conflicts than the AM [27] (again, relaxing inclusion helps COMA). The node miss ratios (Table 4) indicate this clearly. This observation is corroborated by the large number of replacements in COMA (Table 5), as data gets shuffled around when the

AMs try to accommodate replicated lines.

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

The hardware of S-COMA is even simpler than CC-NUMA and the local memory hit access is just as fast. However S-COMA’s consumption of memory is prohibitive for applications with fine-grain sharing because of fragmentation caused by the coarse grain of allocation in the page cache. As a result, under finite memory constraints, the node miss rate and page faulting overhead soar. Table 6 shows the average number of memory misses between two page faults. This number is indicative of fragmentation, being an upper bound on the number of blocks touched during the lifetime of a page frame. Except for Ocean and LU, there is obvious fragmentation. By reducing the usable capacity of the main memory cache, fragmentation directly increases the node miss ratio. The page faulting overhead is very significant in some cases. (Two previous studies either assumed a very low penalty for a page fault [21] or very low memory pressure [20].) There is no real indication from our results that S-COMA is effective at dealing with fine-grain sharing in the general case, unless large amounts of memory can be used without concerns for efficiency (Ocean and LU have coarse grain sharing and would probably perform well on a pure software DSM). In fact, even with twice as much memory, S-COMA does not seem to be a clear winner against CC-NUMA with good data placement.

Due to the reduction of remote capacity misses, RC-NUMA is always an improvement over CC-NUMA. Additionally, it solves some of its technical problems. The RDC

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

Overall, COMA shows consistently good performance across benchmarks and data placement strategies and is only narrowly beaten by other machines when applications exhibit coarse-grain sharing and high spatial locality, mostly because of the slower memory access time on a hit. The hardware cost is somewhat higher because of the AM controller and the tag storage. We note that COMA is much more stingy in memory consumption and less sensitive to memory pressure than S-COMA. Even at 75% pressure, COMA does very well.

Table 3: Cache miss ratios (%) for a 16KB 4-way set-associative SLC

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

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

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

Table 5: Write-back or replacement ratios (per 10⁶ references) (Best placement)

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

Table 6: Average number of block faults (RD/WR misses) between page faults in S-COMA (Best placement)

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

Table 7: Page replication factor in S-COMA

(ratio between the number of allocated page frames in S-COMA with infinite memory and CC-NUMA)

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

6. RELATED WORK

A hybrid CC-NUMA was first prototyped as part of the Alewife project[2]. The simple features of the protocol were supported by hardware and the infrequent, but complex, situations handled in software. Grahn and Stenström later proposed software-only directory protocols [8] for CC-NUMAs with extensive hardware assistance from a custom node controller. Support for fine-grain sharing in software DSM is emphasized in the Stanford FLASH [14], Blizzard [23], Typhoon-0 [20], Typhoon [19], START-NG [3], Blizzard-S [23], and Shasta [22].

Stenström et al. [25] indicate that COMA outperforms CC-NUMA for eight SPLASH benchmarks in the context of hardware DSMs. We have shown that, with hybrid implementations, the performance gap between CC-NUMA and COMA increases and that COMA's more complex coherence protocol is no longer a disadvantage.

Zhang and Torrellas compared hardware COMA and RC-NUMA [27]. Using round-robin data placement for seven SPLASH-2 benchmarks, they show mixed results. However, we find COMA at an overall advantage over

RC-NUMA for round-robin data placement because of the better node hit ratio of COMA.

Saulsbury et al. [21] analyzed the performance of S-COMA for three SPLASH benchmarks concluding that it performs comparably to COMA and CC-NUMA. The study, based on analytical modeling, was biased by several assumptions: relatively small data sets, a fairly large cache (64KB), and unrealistic costs for page faults and COMA replacements. A more realistic study [6] points out the same problems with S-COMA that we have shown here.

7. CONCLUSION

In hybrid DSM systems coherence among nodes is maintained in software with hardware support for fine-grain sharing. This paper makes the first comparative performance evaluation of four hybrid architectures: CC-NUMA, RC-NUMA, S-COMA, and COMA, by using execution-driven simulation of detailed implementations on a common platform. We show that COMA achieves the best (or close to the best) performance level across various applications with widely different behaviors. In particular, the performance level of COMA is independent of the data placement and is very good across various memory pressures. This is in contrast to CC-NUMA and S-COMA where performance exhibits large variations across applications and, thus, the programmer must be aware of the memory organization in order to attain acceptable performance levels. RC-NUMA sometimes performs marginally better than COMA under the best data placement, but is more sensitive to data placement and memory pressure.

As seen in our study and other evaluations of hardware S-COMA [6], coarse allocation in main memory caches may defeat the purpose of supporting fine-grain sharing because of overheads induced by fragmentation. Thus, we advocate to either avoid replication by implementing a hybrid CC-NUMA, or better yet, to replicate memory in units of coherence blocks -- not pages-- and manage (part of) the memory as a set-associative cache by implementing a hybrid RC-NUMA or COMA. COMA and RC-NUMA

require identical hardware resources to support the access and management of the set-associative RDC/AM.

Compared to hardware DSM systems, which are trying to balance access latency, miss rates, and implementation complexity aiming at a high-performance, yet cost-effective and timely design, hybrid systems have a single measure of value: the overhead caused by handling node misses in software. This shifts the preference toward the more complex memory organizations and coherence protocols, which maximize node hit ratios, even at a small penalty for the access latency.

8. REFERENCES

- [1] M. Blumrich et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 142-153, April 1994.
- [2] D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 314-324, April 1994.
- [3] D. Chiou et al. StarT-NG: Delivering Seamless Parallel Computing. *Proc. of the First International Euro-Par Conference*, pages 101-116, August 1995.
- [4] M. Dubois, J. Skeppstedt, and P. Stenström. Essential Misses and Data Traffic in Coherence Protocols. *Journal of Parallel and Distributed Computing*, (29)2:108-125, September 1995.
- [5] 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 International Symposium on Computer Architecture*, pages 144-155, May 1993.
- [6] B. Falsafi and D.A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229-240. June 1997.
- [7] A. Gefflaut, A. Moga, J. Jeong, and M. Dubois. Design and Evaluation of a Software-Controlled COMA. Technical Report 96-03, EE-Systems, Univ. of Southern California, January 1996.
- [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*. pages 38-47, June 1995.
- [9] E. Hagersten, A. Landin, and S. Haridi. DDM--A Cache-Only Memory Architecture. *IEEE Computer*, (25):9, pages 44-54, September 1992.
- [10] M. Horowitz, M. Martonosi, T.C. Mowry, and M.D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. *Proc. of the 23rd Annual International 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 Cache Coherence for Large Scale Multiprocessors. *Proc. of the 1st Symposium on High-Performance Computer Architecture*, pages 286-295. January 1995.
- [13] J. Kubiatiowicz, 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*, pages 274-284, October 1992.
- [14] J. Kuskin et al. The Stanford FLASH Multiprocessor. *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 302-313, April 1994.
- [15] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. *Proc. of the 1988 International Conference on Parallel Processing*, volume II, pages 94-101, August 1988.
- [16] A. Moga, A. Gefflaut, and M. Dubois. Hardware versus Software Implementation of COMA. *Proc. of the 1997 International Conference on Parallel Processing*, pages 248-256, August 1997.
- [17] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. *Proc. of the 1995 International Conference on Parallel Processing*, volume I, pages 1-10, August 1995.
- [18] 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.
- [19] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-Level Shared Memory. *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 325-337, April 1994.
- [20] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. *Proc. of the 23rd Annual International Symposium on Computer Architecture*, pages 34-43, May 1996.
- [21] A. Saulsbury, T. Wilkinson, J. Carter and A. Landin. An Argument for Simple COMA. In *Proc. of the 1st Symposium on High-Performance Computer Architecture*, pages 276-285, January 1995.
- [22] D.J. Scales, K. Gharachorloo and C.A. Thekkath. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174-185, October 1996.
- [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*, pages 297-306, October 1994.
- [24] P. Stenström. A Survey of Cache Coherence Scheme for Multiprocessors. *IEEE Computer*, (23)6:12-24, June 1990.
- [25] P. Stenström, T. Joe and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 80-91, May 1992.
- [26] S. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 24-36, June 1995.
- [27] Z. Zhang and J. Torrellas. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. *Proc. of the 3rd Symposium on High-Performance Computer Architecture*, February 1997.
- [28] Adrian Moga. Design and Performance of the Software-Controlled COMA. Ph.D. dissertation. Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA. May 1998.
- [29] X. Qiu and M. Dubois. Tolerating Late Memory Traps in ILP Processors. *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 76-87. May 1999.
- [30] L. Iftode and J.P. Singh. Shared Virtual Memory: Progress and Challenges. *Proc. of the IEEE*, pages 498-507. March 1999.