An Argument for Simple COMA

Ashley Saulsbury, Tim Wilkinson*, John Carter†
and Anders Landin

‡Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden

Abstract

We present design details and some initial performance results of a novel scalable shared memory multiprocessor architecture. This architecture features the automatic data migration and replication capabilities of cache-only memory architecture *(COMA) machines, without the accompanying hardware complexity. A software layer manages cache space allocation at a page-granularity - similarly to* distributed virtual shared memory *(DVSM) systems, leaving simpler hardware to maintain shared memory coherence at a cache line granularity.*

By reducing the hardware complexity, the machine cost and development time are reduced. We call the resulting hybrid hardware and software multiprocessor architecture Simple COMA*. Preliminary results indicate that the performance of Simple COMA is comparable to that of more complex contemporary all-hardware designs.*

1 Introduction

In this paper we present some preliminary performance results for a scalable shared memory multiprocessor architecture we are investigating. The goal of this architecture is to combine many of the performance advantages of other contemporary shared memory multiprocessor architectures while avoiding their accompanying problems. The result is a hybrid software and hardware *cache-only memory architecture* (COMA) machine that can be built using standard off-the-shelf components. Much of the functionality normally found in COMA hardware has been moved into software, thus simplifying the machine design and reducing development time. Preliminary results indicate that despite the reduced hardware complexity, the performance of the resulting architecture is comparable to that of more complex all-hardware designs. We call this design *Simple COMA*.

Possibly the most popular large scale shared memory multiprocessor design is the *cache coherent non-uniform memory access* (CC-NUMA) architecture, such as embodied by the Stanford DASH [11]. In a CC-NUMA, the machine's physical memory is distributed across the nodes in the machine, but every node can map any page of physical memory (local or remote). When a memory access misses in the local cache, the required data is fetched, according to its physical address, from either the local memory or a remote node's memory. However, the amount of remote data that can be replicated locally, and thus accessed efficiently, is limited by the size of a node's cache, which exacerbates the need for large and expensive caches.

In a *cache-only memory architecture* (COMA [20]), such as the SICS DDM [7] and the KSR-1 [4], the machine's memory is again distributed across the nodes in the machine. However, instead of being allocated to fixed physical addresses, each node's memory is converted into a large, slow[1] cache (or *attraction memory*) by additional hardware. This allows greater and more flexible data replication, but requires more complex hardware and data coherence protocols.

Distributed Virtual Shared Memory (DVSM) systems, such as IVY [12] and Munin [5], forgo all hardware support for coherence management. A DVSM system can exhibit COMA-like properties by migrating and replicating pages from node to node while still keeping them coherent. However, DVSM systems are traditionally page based and rely on the processor to handle coherence management, which can lead to performance problems. Since DVSMs are implemented in software, some of these problems can be mitigated with more sophisticated coherence mechanisms.

Simple COMA divides the task of managing the global virtual address space between hardware and software to provide the automatic data migration and greater data replication of a traditional COMA with much less complex hardware and coherence protocols. As in a traditional COMA, the local memory of a Simple COMA node is used to cache that part of the global

*Permanent address: Systems Architecture Research Centre, City University, Northampton Square, London, UK

†Permanent address: University of Utah, Department of Computer Science, 3190 MEB, Salt Lake City, UT 84112

‡e-mail: {ans,tim,retrac,landin}@sics.se

[1] Traditionally the attraction memory is built using DRAM rather than SRAM, although the state bits needed to maintain coherence are often implemented in SRAM.

virtual address space that local processes are using. However, in a Simple COMA, space in this DRAM cache is allocated and deallocated by the operating system in page-sized blocks using the processor's memory management unit (MMU). This has some similarity to a DVSM system, however unlike in a DVSM system, external hardware is used to maintain coherence at a cache line granularity within the page-sized blocks. This hardware enables the machine to attain the kind of performance one would expect from an all-hardware design, while the operating system becomes responsible for managing the attraction memory space, which greatly simplifies the coherence protocols and helps avoid the need for much of the hardware complexity of conventional COMA machines.

In Section 2 we describe the three conventional architectures (CC-NUMA, COMA, and DVSM) in more detail and discuss their desirable and undesirable characteristics. In Section 3 we describe Simple COMA and examine its potential advantages and disadvantages. In Section 4, we discuss the management of the Simple COMA node memory, and in particular the considerations and impact on hardware design of page replacement. Section 5 contains the results of preliminary simulation studies of Simple COMA. These results indicate that Simple COMA is a viable candidate for building real shared memory multiprocessors. Finally, in Section 6 we summarize our design ideas, discuss the consequences of the preliminary performance results, and draw conclusions.

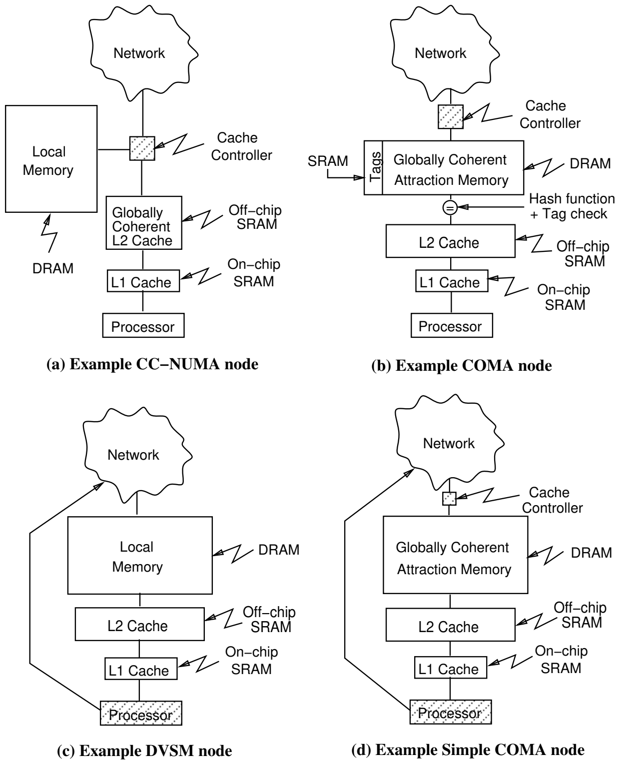

Figure 1: Scalable Shared Memory Architectures

2 Existing architectures

In this section we examine the CC-NUMA, COMA, and DVSM methods of building scalable shared memory machines in more detail, and compare their strengths and weaknesses.

2.1 CC-NUMA

In a Cache-Coherent Non-Uniform Memory Access Machines (CC-NUMA) machine (see Figure 1a), physical memory is distributed across the processors so that a small portion of the memory is local to each. The remainder must be accessed remotely across an interconnection network. The access performance of the local memory is similar to that of a uniprocessor, while the latency to access data in a remote memory can be an order of magnitude or more larger, hence the term *non-uniform memory access.*

As on uniprocessors, caches are used to improve performance. Figure 1a illustrates the case where there is both a small on-chip (L1) cache and a larger off-chip (L2) cache, both constructed from fast but expensive SRAM. These caches can contain data whose home is in the local memory or data that was accessed from across the network. The memory system hardware must keep cached copies of data coherent throughout the machine. In a CC-NUMA this is done by the cache controller (shown in the shaded box[2] in Figure 1a), which is generally implemented using a directory-based protocol [1] and some SRAM state.

In this design, data from remote nodes can only reside locally in a node's caches, never in its main (DRAM) memory. This feature restricts the total amount of data that each node can replicate from remote nodes at any given instant. Therefore, the performance of CC-NUMA machines is severely limited if data is incorrectly placed at physical memory locations away from the nodes that use it most, if data needs to migrate from one set of nodes to another, or if an application requires a large amount of data to be replicated on each processor. This effect leads to pressure when designing CC-NUMAs to build very large (and expensive) second level caches.

The addition of operating system managed *page migration* [3] allows the designated "home" of a data page (the node where the corresponding physical memory resides) to be moved. This goes a little way towards solving the problem of incorrect data partitioning. Additionally, pages can be replicated manually, but this greatly complicates coherence management. In general, these mechanisms are slow and complex, work only at a page granularity, and have not been shown to improve performance significantly beyond intelligent

[2]In Figure 1, shading is used to highlight where each architecture's control logic is situated.

initial page placement [10].

2.2 COMA

In a Cache Only Memory Architectiure (COMA) machine, additional hardware is used to implement a large DRAM cache (or *attraction memory*) instead of the large local memory of a CC-NUMA. For a COMA such as the SICS DDM [7], this extra hardware consists of: (i) a hashing function to map from the physical address used by the processor to the real physical address of the set of DRAM cache lines that might hold this entry, (ii) tag memory and comparators that determine which cache line in the designated set contains the desired data (if any), and (iii) extra state memory to maintain the state of a selected cache line (see Figure 1b). The purpose of this hardware is to decouple the physical address generated by each processor from the "home" location of the data, as opposed to in a CC-NUMA, where the physical address of a data item specifies on which node that data item's "home" resides.

The advantage of COMA over CC-NUMA is the flexibility provided by this additional hardware. Data is automatically copied or migrated to physical memory (DRAM) on the node(s) where it is being used (thus the term *attraction memory*). This dynamic rebalancing of the data held on a node means that COMA can exhibit all of the properties of CC-NUMA, including those of a CC-NUMA with page migration or replication, for CC-NUMA tailored applications. In addition, COMA machines can run applications that do not map well to CC-NUMA architectures [6], such as applications that have a per-node working set larger than the size of each node's cache and applications with dynamic data access patterns in which data cannot be statically partitioned across the physical memories effectively.

Unfortunately COMA's flexibility requires nonstandard memory system hardware, introducing a price-performance tradeoff (more hardware leads to higher performance, but also higher cost and longer development times). A hash function with dedicated tag memory must be placed between the L2 cache and local memory to convert the "global" physical addresses generated by the processor to "real" physical addresses recognized by the local node's memory. Furthermore, the coherence protocol implementation is complicated by the need to preserve the last copy of a data item; unlike a CC-NUMA there is no reserved home for a cache line that is evicted from a node's attraction memory, so care must be taken not to throw away the last copy of a cache line [9, 21].

2.3 DVSM

Distributed Virtual Shared Memory (DVSM) systems also provide a shared memory abstraction, but memory coherence is implemented entirely in software (see Figure 1c). Like in COMA, in a DVSM system a node's main memory is used as a local cache of the machine's global virtual address space. Traditionally, DVSM systems use the processor's MMU to detect accesses to shared data that is not present on the local node and must be fetched across the network, or that is currently mapped read-only as part of the coherence management mechanism. When such an access occurs, the DVSM runtime system is invoked to perform the necessary operations to load the data or maintain coherence.

There are two major advantages to software DVSM systems. As they are implemented entirely in software, they can employ sophisticated algorithms to manage memory and adapt to changing memory access patterns. Even early DVSM systems like IVY [12] exhibited many of the desirable properties of COMA, e.g., automatic data migration and replication. In addition, DVSM systems have very simple hardware requirements. and thus can operate on anything from tightly coupled distributed memory multiprocessors like the CM-5 [14] to a network of workstations [5, 12].

However, these advantages of DVSM systems come at a potentially serious cost in terms of performance. Because all coherence management is done in software, the overheads associated with DVSM are high. Handling page faults in software can be several orders of magnitude more expensive than handling a cache miss in hardware, especially on CPUs with imprecise exceptions. Secondary effects of handling a page fault, such as polluting the caches with OS data, decrease application performance. Also, the processor is interrupted when coherence management requests (e.g., load and invalidation requests) from remote nodes need to be handled. These requests have nothing to do with the operation of the application on the processor concerned, yet they interrupt the processor and disturb its caches and program flow as much as, if not more than, its own coherence management operations.

In addition to fault handling overheads, there is typically a significant communication protocol overhead for each coherence management message, which can dominate the actual wire-time. This overhead is increasing as network bandwidths improve. The problem can be mitigated by introducing a dedicated network technology, as was done in the Meiko CS-2 [13], or using delayed consistency to reduce the number of messages sent, as was done in Munin [5], at the cost of more hardware or more complex coherence protocols.

Finally, DVSM systems are particularly sensitive to *false sharing*, wherein unrelated data items that happen to reside on the same page can interfere with one another because the MMU provides support only at the granularity of a page. This problem can be addressed by using a relaxed memory consistency protocol, such as that in Munin, at the expense of further polluting the processor cache (to copy or compare data to handle fine-grained sharing), or by reducing the granularity of coherence. The latter requires hardware support not available on most platforms, unless, such as in the Wisconsin Wind Tunnel [14], or special hardware, such as in Typhoon/Tempest [15].

3 Simple COMA

In this section we describe in more detail the Simple COMA idea [8, 16] and examine various aspects of the architecture that allow a simpler hardware architecture than COMA or CC-NUMA, but without compromising the performance.

3.1 Building an attraction memory

In a uniprocessor, the operating system manages physical memory as a cache of the much larger virtual memory working set. In a multiprocessor built from conventional microprocessors, similar techniques can be used to cache parts of the global virtual address space in each node's local physical memory - as is also done by DVSM systems. When space is required by an application for new data or shared data that needs to be fetched, software allocates pages in the local physical memory and inserts the appropriate virtual to physical address mappings into the MMU. If we consider a node's memory to be an attraction memory, then each time a processor makes a memory access, its MMU effectively performs the attraction memory tag check by looking for a page mapped at the given virtual address. If a mapping exists, then space has been allocated for the data in the local (attraction) memory. If not, space has not been allocated and the MMU generates a VM exception so the processor can map in the data.

Managing the attraction memory in software has several advantages. Since any virtual page can be mapped to any physical page, the attraction memory is fully associative. In addition, the MMU performs the data tag check in Simple COMA, so no dedicated tag memory is required other than the existing page table, unlike in a traditional COMA. As well as simplifying the hardware, this avoids the delays in accessing data stored in the local attraction memory that occur in a traditional COMA while it searches the appropriate cache slots to determine which one contains the data (assuming a set associative attraction memory). This set lookup can introduce a particularly heavy penalty if, as in the KSR-1, the slots are checked linearly. Avoiding this overhead for local memory accesses is clearly important since the purpose of the COMA cache (attraction memory) is to hide the latency of remote memory accesses.

3.2 Maintaining coherence

While Simple COMAs allocate and deallocate memory at a page granularity, as in a DVSM system, data is kept consistent entirely in hardware, at a fine granularity, and without processor intervention. Each page in a node's attraction memory is sub-divided into a number of *coherence units*[3], and associated with each such unit in a shared page is a small amount of state information. A protocol engine connected directly to the communications network keeps these coherence units consistent.

The order of events for a read in a Simple COMA is as follows:

1. The MMU checks to verify that the (global) virtual address being referenced is mapped locally. If it is not mapped, a page fault occurs and the operating system must map the page, as detailed in Section 4, and resume the access.

2. After the MMU check succeeds (or in parallel with the check in the case of a virtual L1 cache), the SRAM caches are examined to determine whether the desired data is present. If it is, the cache supplies the data and the access completes.

3. If no SRAM cache contains the data, the local (attraction) memory is read. In parallel with the read, the state information associated with the coherence unit accessed is checked to verify that the data in the local attraction memory is valid. If so, the local attraction memory supplies the data and the access completes. If the data in the local attraction memory is not valid, the external memory management hardware takes steps to acquire a copy of the coherence unit.

Writes are handled analogously.

Note that it is possible for the desired data to be mapped by the MMU but not be valid in the local attraction memory. The coherence management hardware must maintain state for each coherence unit in each shared page mapped into the local attraction memory. In a sequentially consistent system this state might indicate:

Invalid – This node does not have a copy of the data, it must be fetched from another node.

Exclusive – This node holds the only copy of the data, it may be freely read or written.

Shared – This node holds one of many copies of the data in the machine. Therefore, the data may only be read. If the memory access is a write, the coherence hardware must take steps to keep the other copies in the machine consistent (i.e., send updates or invalidates).

Pending – This may be one or more states, indicating that the state of the data is in flux (i.e., data

[3] Although coherence units are essentially cache lines, we use a distinct term to avoid implying that the attraction memory is made of fast "cache" memory.

is being fetched for a read miss but has not yet arrived).

The separation of space allocation, performed by software, from coherence management, performed by hardware, enables the cache coherence protocols used in a Simple COMA machine to be simplified significantly compared to those used in both traditional COMA and CC-NUMA machines. In both traditional COMA and CC-NUMA, the hardware cache controller must deal with replacing cache lines from the local node. In the case of CC-NUMA, if a cache line is being evicted due to contention, the associated directory must be updated to indicate that the node no longer holds a copy of that data, and the data must be written back if it has been modified. In traditional COMA, the cache controller also must handle contention in the attraction memory. This is harder than dealing with contention in a CC-NUMA cache because there is no reserved "home" memory location for the data. Instead, when evicting the last copy of a cache line, the hardware must find a "home" in another node's attraction memory. This is perhaps the most complex feature of traditional COMA hardware. Simple COMA does not have this problem at the coherence unit level because space is always reserved for a cache line, (even if the data is not present), until the entire page is replaced from the attraction memory under the coordination of software. Thus, the Simple COMA hardware cache coherence protocol does not need to handle arbitrary cache line replacement – it only needs to provide simple support for handling evictions during page replacement.

3.3 Global addressing

There must be a way to uniquely identify each shared data item so that the cache controllers on different processors can communicate the identity of a piece of shared data that they need to manipulate (e.g., load or invalidate) to remote processors.

In both CC-NUMA and traditional COMA architectures, the physical address generated by the processor's MMU is used as the global identifier for a piece of data in the machine. In Simple COMA, however, the physical address only indicates the location of the data in the local attraction memory. Since each node's attraction memory is fully associative, the location of data in one node need have no relationship to the location of a copy of the same data in another. Therefore, some other unique identifier is required to specify data items within a Simple COMA machine, and a mechanism must be provided to translate physical addresses to and from these unique identifiers when communicating with other nodes. This translation mechanism is implemented by the network cache controller. Note that the data's virtual address does not suffice as a global identifier, as system calls like `mmap()` allow the same data to be mapped to different virtual addresses in different processes. [4]

One method of converting local physical addresses to global identifiers when initiating a request from the local node to remote nodes is to use a translation table with one entry per local physical page. This table is not unlike the page table used by the MMU, and can also be held in normal memory. To convert global identifiers to local physical addresses, (which is necessary to locate the data and state associated with global identifiers found in incoming network requests), a sparse translation table holding one entry per global identifier that has a local physical address mapping can be employed. It is important to note that there need be only one entry in each of these tables per shared page, not per coherence unit, because, while coherence takes place at a fine grain, sharing takes place a page level. Hence, a global identifier specifies a page and an index within this page is used to find the appropriate coherence unit. These conversions can be optimized, for example by including a fully associative translation cache (TLB) in the cache controller.

4 Attraction memory management

Simple COMA performs the most complex aspect of COMA operation, the allocation and replacement of space in the attraction memory, in software. While this simplifies the COMA hardware, it necessarily increases the cost of replacing data in the attraction memory. However, because replacement is managed in software, it is possible to implement more sophisticated replacement algorithms, even allowing direct interaction with applications to manage the local working set efficiently.

Managing a Simple COMA node's attraction memory can be divided into two basic levels. The lower level handles the direct interaction with the hardware, coordinating the allocation and deallocation of globally shared pages. The upper level is responsible for determining which pages should be replaced when space for new data is needed in the attraction memory and for ensuring that the last copy of a page is not discarded inadvertently. There are a large number of page replacement algorithms that could be employed, and a complete discussion is beyond the scope of this paper. However, page replacement is an important component of a Simple COMA system, so we will discuss various aspects of the problem in this section and present the results of a preliminary evaluation of its impact on performance in Section 5.

4.1 Page allocation

If an application attempts to access data for which no space is allocated in its local attraction memory, the processor's MMU generates a page fault. The page

[4] Unless a Single Address Space OS is used.

fault handler must allocate a free page in the processor's local memory and initialize a page table entry in the MMU, just like the page fault handler in a conventional operating system. The fault handler also must initialize the state of each of the coherence units within the page. If the page being loaded is not replicated, its coherence units can simply be set to enable the local processor full read and write access. If, however, the page is replicated, the state of each of the coherence units should be initialized to *invalid*. Finally, the forward and reverse physical address to global identifier translation tables should be initialized.

4.2 Page replacement

Deallocating a page from the local attraction memory is slightly more complex than allocating one. First, the mapping for the virtual page must be removed from the page table and the associated TLB entry flushed, if necessary. Then, the valid data contained in the freed physical page must be flushed or invalidated. Consistency units marked as Invalid can simply be ignored. Similarly, the data corresponding to coherence units marked as Shared can simply be ignored, although the coherence directory must be informed to ensure that if only one other copy of the data remains, it becomes Exclusive. Coherence units marked as Exclusive must be transferred to another node that is sharing the page. If other nodes are using the page, the higher level page management code will be able to identify at least one node that still has a copy of the page and initiate a transfer of the exclusive coherence units to that node. Unlike a traditional COMA, transferring a coherence unit to another node cannot force further replacement on that node [9], since in Simple COMA that node will already have the space allocated for the coherence units if it is sharing the page.

As replacement is being controlled by software it is easier to ensure that the final two copies of a page are not removed simultaneously, a situation that must be explicitly addressed in the hardware coherence protocol of a traditional COMA machine. Depending on the replacement protocol and its implementation, the Simple COMA hardware could either immediately remove the current node from the list of nodes sharing the page, or do nothing at all and ignore subsequent invalidation requests involving the discarded page. In the latter case, the high level page management software must designate an "owner of last resort" for each page that cannot silently discard the page without designating a new "owner of last resort." Finally, if the list of coherence unit copies is distributed with the data in a coherence protocol design, as in the Stanford Distributed Directory protocol [19], then the coherence units must be unhooked from the linked lists before the page can be reallocated.

If the page chosen for replacement is not being shared by another node, the operating system can either write the data out to disk or persuade another node to take the page. The latter will often be the preferred option should the page be needed later on the same node. If paging to disk is the preferred option, it is much easier to do in a Simple COMA than in either a CC-NUMA, which must persuade all the other nodes sharing the page to flush their caches, or a traditional COMA, which must not only attract the various cache lines to the local node's memory but also persuade the hardware to keep the data there while paging takes place. In Simple COMA, if a page is exclusive to a node then the data is already collected and valid, and thus may be paged from the node to disk just as in a uniprocessor.

4.3 Caches and attraction memory

For performance reasons, any Simple COMA machine (or other multiprocessor) is likely to have multiple levels of cache between the attraction memory and processor in each node. These caches may be any combination of write through or write back, virtual or physical. In the case of a node employing write-back caches, when moving exclusive coherence units to another node during page replacement, any dirty cache lines corresponding to the page being replaced must be written back and invalidated from the cache. Similarly, coherence units in the shared state, within either write-back or write-through caches, must also be invalidated if present in higher level caches. This action prevents stale data in the caches from accidentally being used if a page is mapped back again - even though the correct data is not actually in the attraction memory.

Since the processor validates all its memory accesses with the MMU, it cannot access data from its caches for which there is no mapping to the local memory. For each valid cache line in the first or second level caches, there must be corresponding space allocated in the node's main memory. Thus, Simple COMA's attraction memory is fully inclusive [2] with respect to the various levels of cache. Due to this, no action is required for invalid coherence units on replacement of their containing page. If the state is invalid in the attraction memory, data cannot be in a higher level cache either. This property reduces the overhead of page replacement.

5 Preliminary results

To evaluate the effectiveness of Simple COMA, we simulated a CC-NUMA machine, a traditional COMA machine, and a Simple COMA. The output of these simulations included the number of hits and misses at various levels of the memory hierarchy, the number of inter-node coherence operations that were required to maintain coherence, and the number of page faults suffered by the applications. This information was fed into an analytical model that calculated the approximate number of machine cycles the computation is stalled for each architecture and the impact of Simple COMA's page replacement overhead on performance.

5.1 Applications

The simulations in this study modeled moderately sized machines running applications chosen from the Splash [18] benchmark suite:

MP3D A simple simulator for rarified gas flow over an object in a wind tunnel. Data: 80000 particles for 40 time steps using the default test geometry.

OCEAN An SOR application modeling the behavior of a small section of ocean surface. Data : 130 x 130 grid with a convergence tolerance of 10^{-7}.

Barnes-Hut A simulation of gravitational body attraction. Data : 10240 bodies in a plummer distribution simulated for 3 time steps.

Each application's data set size is approximately 4 Mbytes. These applications are all barrier-synchronized step-scheduled applications. Thus, their performance is not highly dependent on various system latencies, as is the case for dynamically scheduled applications. Furthermore, these applications all exhibit quite bad spatial locality at the page granularity (particularly MP3D and Barnes-Hut), and thus represent serious challenges for Simple COMA and make our results fairly conservative.

5.2 Simulation Model

For each experiment we simulated a 16-node multiprocessor with a single DEC Alpha processor per node. Each processor was configured with an 8-kilobyte first level cache and a 64-kilobyte second level cache. We chose to simulate fairly small caches to compensate for the fact that the working set sizes are also quite small. The first level cache was modeled as direct-mapped and write-through, while the second level cache was modeled as two-way set associative and write-back. We used 64-byte coherence units throughout the memory hierarchy (L1-cache, L2-cache, and attraction memory).

To examine the behavior of each architecture as a function of the fraction of total machine memory filled by the application, we ran each simulation with varying amounts of total simulated machine memory. A simple sequentially consistent protocol was simulated for each architecture, and we modeled only a trivial point to point contention-free network.

For a traditional all-hardware COMA, a **fully associative** attraction memory was simulated. Such an attraction memory is not easily implemented in hardware, but we chose to simulate this rather than make an arbitrary decision on the number of ways of associativity. Thus, our results for traditional COMA may be somewhat optimistic. A random replacement strategy was used to select cache lines for eviction. For the applications chosen, random replacement produced better results than a least recently used strategy.

Operation	CC-NUMA	Trad. COMA	Simple COMA
Load			
1st Lvl from 2nd lvl	10	10	10
2nd lvl from local mem	50	50	50
Remote	300	300	300
Store			
Avg penalty to 2nd lvl	1	1	1
2nd lvl to local mem	50	50	50
Remote	450	450	450
Trad COMA			
Tag check	—	20	—
Replacement	—	600	—
Page Fault	150	150	200

Table 1: Cycle penalty parameters for operations in the analytical model

A trivial random replacement strategy was also modeled for Simple COMA, except that only pages shared with other nodes were allowed to be evicted. Thus, pages unique to a node can only leave that node if another node accesses data in the same page.

5.3 Analytical model

Using information about the type and number of cache misses for each target architecture and reasonable estimates for the cost in lost computing cycles for each type of miss, we can derive the approximate number of cycles spent waiting on memory operations for each of the architectures. We assume that all three architectures use the same memory technology, on-chip first level caches, off-chip SRAM second level caches, and DRAM for the main node memory (even for the COMA attraction memory). The parameters used by our analytical model are given in Table 1.

These parameters are in keeping with current memory system technology. We assume that the first level caches are write-through, so we allow an average store penalty to account for stores that block waiting for previous stores to complete. For traditional COMA, the attraction memory tags must be checked after each L2 cache miss to determine if the required data is in the attraction memory. This cost must be paid even if the data is not present in the attraction memory. In an aggressive COMA implementation (e.g., the SICS DDM), the tag memory is implemented in SRAM, so the tag check penalty is less than the cost of a full DRAM memory cycle.

There is no direct page replacement cost for the Simple COMA, although the indirect cost of removing cache lines contained in the page being evicted from the L1 and L2 caches was modeled. Page replacements occurred sufficiently infrequently that it

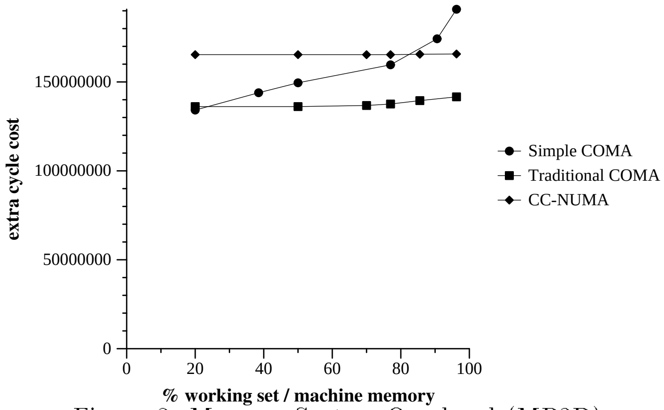

Figure 2: Memory System Overhead (MP3D)

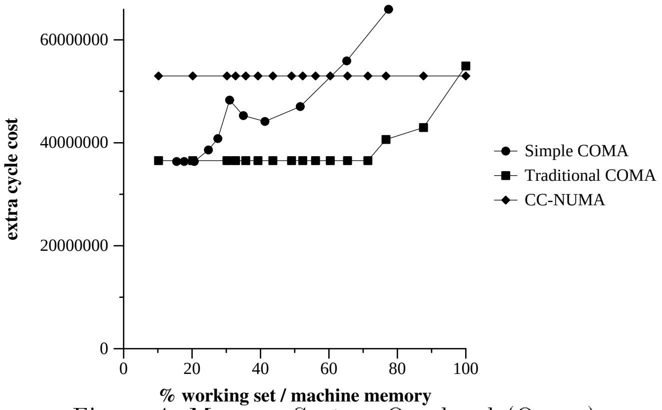

Figure 4: Memory System Overhead (Ocean)

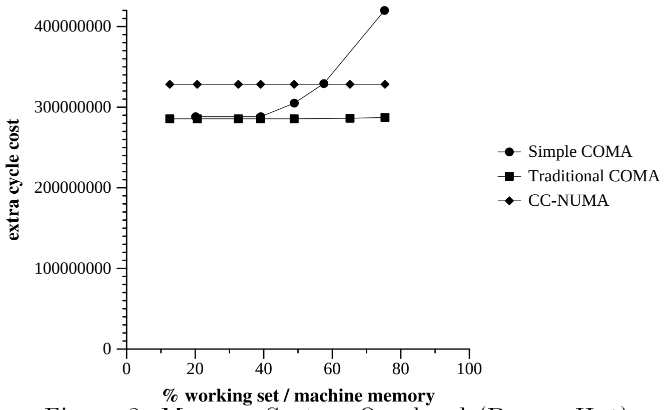

Figure 3: Memory System Overhead (Barnes-Hut)

would be relatively easy to maintain a small pool of free pages, thus allowing the page being loaded to be immediately mapped (and thus accessed). Then, the page being replaced can be evicted by the coherence hardware while the computation continues.

A full description of the analytical model used in this study can be found elsewhere [17].

5.4 Simulation results

Figures 2 through 4 illustrate the estimated extra number of machine cycles required to handle first and second level cache misses and inter-node coherence-related communication for a CC-NUMA, a traditional COMA, and a Simple COMA machine. For each series of tests, we kept the problem size constant and varied the size of physical memory in the machine (i.e., machine memory decreases and the fixed problem size becomes an increasingly large percentage of the total machine memory as you move from left to right). For all of the simulations, the application data set fits into the aggregate machine physical memory.

5.4.1 CC-NUMA

When the percentage of aggregate physical memory used to hold the application working set is small, Simple COMA causes up to 33% fewer delay cycles than CC-NUMA. This performance improvement is directly attributable to the COMA architectures' ability to replicate more data locally than the CC-NUMA. However, as the amount of free memory available for data replication decreases, the performance of Simple COMA lags compared to CC-NUMA.

For CC-NUMA, the extra cycle cost of shared data accesses is independent of the fraction of main memory in use. The memory management overhead is simply a function of the size of the L1 and L2 caches and the size of the application data set. Physical memory not filled with problem data (or code) is unused. This illustrates both a strength and a weakness of CC-NUMA. To their advantage, CC-NUMA machines are relatively insensitive to the fraction of their main memory that is full, as long as it is below 100% and thus paging to disk is avoided. However, CC-NUMA's insensitivity to physical memory size comes at a price. If the amount of cache per node is held constant, CC-NUMA machines will tend to scale poorly as extra nodes are added to the machine and the problem size is increased proportionately. The reason for this is that as the problem size increases, the number of L2 cache misses on each node will increase because of the limited capacity in the L2 cache, unless the data is perfectly partitioned. The extra physical memory added to the machine by the addition of new nodes cannot compensate for the higher L2 cache miss rate, because CC-NUMA does not migrate data to physical memory on nodes where the data is being used. The only solutions to this scalability problem are to increase the size of each node's L2 cache, which is significantly more expensive and complex than adding DRAM, or to require software to carefully manage physical memory to perform COMA-like replication.

5.4.2 Conventional COMA

Traditional COMA generally outperforms CC-NUMA significantly since it can use the free memory in the machine for replicating the applications data set. This trend holds until the application data set comes close to 100% of the machine memory, at which point the loss of replication space begins to impact traditional COMA performance. Unlike CC-NUMA, traditional COMA architectures follow the inclusion property for the L1 and L2 caches to avoid adding yet more hardware complexity. If, however, we were to break the inclusion property for traditional COMA, in the limit where the application data set fills the main machine memory, we would see exactly CC-NUMA behavior. By breaking the inclusion property, we can make a traditional COMA machine that should perform no worse than CC-NUMA, but that can perform significantly better if there is spare memory available for replication.

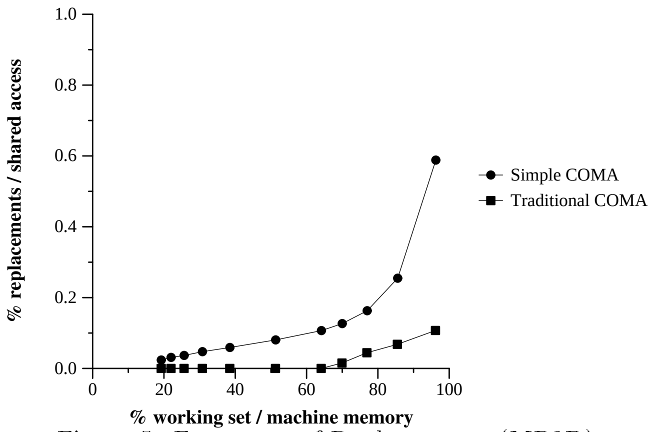

Figure 5: Frequency of Replacements (MP3D)

5.4.3 Simple COMA

When the percentage of aggregate memory used to hold the application working set is small, and thus there is a lot of free memory available for replication, Simple COMA achieves performance essentially identical to traditional COMA. In this case, attraction memory fragmentation caused by Simple COMA allocating it in page-sized blocks is minimal. As the amount of free memory decreases and contention for the available attraction memory increases, Simple COMA's performance drops off more quickly than traditional COMA's due to the aforementioned fragmentation. Simple COMA significantly outperforms CC-NUMA when there is sufficient free memory, but as the amount of free memory decreases, Simple COMA's performance eventually becomes worse than CC-NUMA's. However, as discussed above, even for applications with large data sets, Simple COMA's performance can be improved with the addition of (cheap) DRAM, which is not the case for CC-NUMA.

The applications studied all have bad spatial locality at page granularity, which hurts the performance of Simple COMA. The effect of this poor page-grained locality is to force Simple COMA to allocate space in the attraction memory that is not entirely used. For example, a page may be allocated to allow access to only two coherence units. While the time taken to perform the page allocation and deallocation is not great, it does have the effect of denying space in the attraction memory to replicate other data - data that in a traditional COMA might remain present.

However, the results are encouraging, given that our simulations used a very simple page replacement algorithm (*random*) and the application datasets simulated were small[5]. For Barnes-Hut and Ocean, Simple COMA performed better than CC-NUMA until the application dataset reached over 60% of the total machine memory, and for MP3D the cross-over point was over 80%. In fact, in MP3D at the 25% level, Simple COMA slightly outperformed the traditional COMA due to the extra access penalty of the tag check hardware in the traditional COMA's attraction memory.

[5] If the total number of pages per node is small, then the relative impact of evicting a single page increases

5.4.4 Simple COMA page replacement

Perhaps the most interesting clue to the performance of Simple COMA is the page replacement behavior. In Figure 5, we see that even when application's dataset fills 80% of the available machine memory only 0.2% of the accesses to shared memory, (6.77% of the attraction memory misses, or 0.08% of the total application loads and stores), cause a page fault. However, the cost of handling a page fault to manage the attraction memory is quite small compared to the overall cost of accesses to other nodes for coherence actions. What is important therefore, is not the page faults, but the effect of allocation in the attraction memory due to the page granularity. When a coherence unit is not found in the attraction memory, an entire page of coherence units must be evicted to make space for the one requested. This can result in the eviction of coherence units that do not directly contend with the required coherence unit, a phenomenon we call *false replacement*.

Since the page replacement algorithm of Simple COMA is implemented in software, greater sophistication can be provided to help counterbalance the effects of false replacement. This has the potential to significantly improve the hit rate in the attraction memory. Information from the application or the operating system is also directly available to help influence this mechanism. For certain applications with good page level locality, it should be possible to out perform a conventional COMA.

6 Conclusions

We have presented a novel scalable shared memory multiprocessor architecture called *Simple COMA* that supports the automatic data migration and replication features of COMA machines, but replaces much of the complex hardware of COMA with a DVSM-like software memory management layer. Unlike DVSM, however, pages are sub-divided into cache-line sized blocks, and the coherence of these blocks is maintained in hardware. By allowing shared data to be replicated in a node's main memory (in addition to its caches), the number of remote memory accesses is greatly reduced compared to traditional CC-NUMA machines. This often results in superior performance for Simple COMA despite its relative hardware simplicity.

Moving the most complex aspect of the COMA protocol, the allocation and replacement of space in the attraction memory, to software simplifies the machine design and reduces development time, which should give Simple COMA a significant advantage in terms of time to market and price. The tradeoff associated with Simple COMA compared to traditional COMA is that Simple COMA allocates the attraction memory in page-sized units, which can lead to worse attraction memory utilization and result in L1 and L2 cache evictions that would not occur in traditional COMA when a page in the attraction memory is replaced. However, our preliminary simulation results indicate that even for problems that have poor page-level locality, the performance of Simple COMA is comparable to that of all-hardware designs despite its much simpler design. Further study of Simple COMA is needed to determine its sensitivity to various systems issues, such as network latency and the performance ratio between the various levels of the memory hierarchy. However, the results of our preliminary study lead us to believe that Simple COMA is a viable multiprocessor architecture for future machines, and thus warrants further research.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, June 1988.

[2] J-L. Baer and W-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, 1988.

[3] W.J. Bolosky, M.L. Scott, R.P. Fitzgerald, R.J. Fowler, and A.L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the 4th ASPLOS Symposium*, pages 212–221, 1991.

[4] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Thirteenth Symposium on Operating System Principles*, October 1991.

[6] E. Hagersten. *Toward Scalable Cache Only Memory Architectures*. PhD thesis, Royal Institute of Technology, Stockholm, 1992.

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

[8] E. Hagersten, A. Saulsbury, and A. Landin. Simple COMA node implementations. In *Hawaii International Conference on System Sciences*, January 1994.

[9] T. Joe and J.L. Hennessy. Evaluating the memory overhead required for coma architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 82–93, 1994.

[10] R.P. LaRowe and C.S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, December 1991.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

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

[13] Meiko Limited. CS-2 Product Description, 1992.

[14] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of ACM SIGMETRICS Conference*, May 1993.

[15] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, 1994.

[16] A. Saulsbury. Supporting fine-grain shared memory. Technical Report from January 1993, now publically available as SICS Tech Report number T94:06, Swedish Institute of Computer Science, January 1994.

[17] A. Saulsbury, T. Wilkinson, J. B. Carter, and A. Landin. Handling replacement in simple COMA. Swedish Inst. of Computer Science Report, 1994.

[18] J.S. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Stanford University, Report, April 1991.

[19] M. Thapar and B. Delagi. Stanford Distributed-Directory Protocol. *IEEE Computer*, 23(6):78–80, June 1990.

[20] D.H.D. Warren and S. Haridi. Data Diffusion Machine–a scalable shared virtual memory multiprocessor. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[21] X. Zhang and Y. Yan. Latency analysis of cc-numa and cc-coma hierarchical ring multiprocessors. In *Proceedings of the 1994 International Conference of Parallel Processing, Vol. I*, pages 174–181, August 1994.