Simple COMA Node Implementations

Erik Hagersten*
Sun Microsystems Computer Corp.
2550 Garcia Av.
Mountain View, CA 94043-1100

Ashley Saulsbury and Anders Landin
Swedish Institute of Computer Science
Box 1263
164 28 KISTA; SWEDEN

Abstract

Shared memory architectures often have caches to reduce the number of slow remote memory accesses. The largest possible caches exist in shared memory architectures called Cache-Only Memory Architectures (COMAs). In a COMA all the memory resources are used to implement large caches. Unfortunately, these large caches also have their price. Due to its lack of physically shared memory, COMA may suffer from a longer remote access latency than alternatives. Large COMA caches might also introduce an extra latency for local memory accesses, unless the node architecture is designed with care.

We examine the implementation of COMAs, and consider how to move much of the complex functionality into software. We introduce the idea of a simple COMA architecture - a hybrid with hardware support only for the functionality frequently used. Such a system is expected to have good performance, and because of its simplicity it should be quick and cheap to develop and engineer.

1 Introduction

Multiprocessors with cache-coherent shared memory can be built in many ways. Systems based on a single bus suffer from bus saturation and therefore typically have only some tens of processors, each with a local cache. The contents of the caches are kept coherent by a cache-coherence protocol, in which each cache snoops the traffic on the common bus and prevents any inconsistencies from occurring [21]. This architecture provides a uniform access time to the whole shared memory, and is therefore called uniform memory architecture (UMA).

In architectures with distributed shared memory, known as Non-Uniform Memory Architectures (NUMA)[1], each processor node contains a portion of the shared memory; consequently, access times to different parts of the shared address space can vary. NUMAs often have networks other than a single bus, and the network delay to different nodes might vary. Early NUMAs did not have coherent caches and left the problem of coherence to the programmer. Research activities today are striving toward coherent NUMAs with directory-based cache-coherence protocols, e.g. Dash [14] and Alewife [1]. Programs can be optimized for NUMAs by statically partitioning the work and data. Given a partitioning where the processors make the most of their accesses to their part of the shared memory, a better scalability than for UMAs can be achieved.

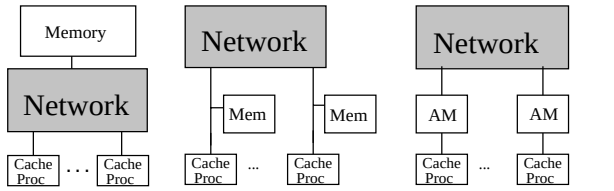

Figure 1: *Comparing COMA to more conventional architectures.*

In a *cache-only memory architecture*, COMA [8], the memory organization is similar to that of NUMA in that each processor holds a portion of the shared memory space. However, the partitioning of data between the memories is not static, since all distributed memories are organized as large (second-level) caches. The task of such a memory is twofold. Besides being a large cache for the processor, the memory may also contain some data from the shared address space that the processor never has accessed, i.e., it is a cache and a virtual part of the shared memory at the same time. We call this intermediate form of memory *At-*

*This work was done while the author was at SICS.

[1] In this paper, NUMAs are assumed to be cache-coherent. Also referred to as CC-NUMA.

traction Memory (AM). A coherence protocol will attract the data used by a processor to its attraction memory. The unit of coherence, called an *item*, is comparable to a cache line, and is moved around by the protocol. On a memory reference, a virtual address is translated into an item identifier. The item identifier space is logically the same as the physical address space of conventional machines, but there is no permanent mapping between an item identifier and a physical memory location. Instead, an item identifier corresponds to a location in an attraction memory, whose address tag matches the item identifier. There are cases where multiple attraction memories could have matching items, i.e., the item is replicated. Examples of such architectures are the DDM [8] and the KSR1 [13].

The new requirements of building efficient large caches have led to the design of a proprietary processor cache in the KSR1, as will be described in more detail later. Other proprietary parts of the KSR1 are the processor and the network. This proprietary choice results in a processor three to four times slower than today's commercial offerings, and remote latencies three to four times longer than necessary. In spite of its longer delays, the KSR1 has proven a performance comparable to the Dash architecture for the SPLASH applications [11]. Another comparative study assuming that a COMA implementation suffer from longer remote access delays also concluded that half of the studied applications were in favor of COMA and half of NUMA [22]. However, since both these studies compares one specific implementation of *a hierarchical* COMA to a NUMA based on a *2-D mesh network*, it is not clear what conclusions can be made on the general NUMA-or-COMA question.

However, the latter of the two studies also compared NUMA and COMA on equal grounds (COMA-F), concluding that COMA beat NUMA for all studied applications. Yet another comparative analytical study of general implementations of NUMA and COMA, covering a large design space, reports a performance advantage for COMA architectures for the same set of programs [4].

The aim of this work is to find a *simple* COMA implementation which is compatible with existing computer technology and knowledge, such as cache-coherence protocols, microprocessor implementations, programming paradigms, and operating systems.

In the remainder of this paper, we first discuss some general issues for COMA architectures; the next section reviews some existing COMA node implementation proposals, followed by a description of our simple COMA proposal. The paper is concluded by a complexity and performance study followed by a summary of related work and our conclusions.

2 COMA properties

This paper does not take a position on network topology and/or choice of coherence protocol. However, a short discussion about these two important topics might be appropriate.

Recent years have seen extensive studies of the problem of maintaining coherence among read-write data shared by different caches—for example directory-based and snooping-based techniques [21].

Even though both COMAs being built today, KSR1 and DDM, rely on a hierarchical network topology, COMAs can be built upon general networks [7, 22]. The cache-coherence protocol for a COMA can adopt the techniques of other cache-coherence protocols [14, 1, 10] and add functionality for finding an item on a cache-read miss and for handling replacement [8]. The search for the item compensates for the lack of a home for data in a COMA. The problem of finding an item on a read miss has already been addressed in the NUMA protocols for situations where a dirty copy of the requested item resides in a node other than the home node. Most accesses missing in COMA's attraction memory at steady state execution are likely to be coherence misses, caused by true or false data sharing between one or more processors. For those misses, chances are high that dirty data will reside in a remote node in a NUMA architecture, i.e., about the same amount of overhead can be expected for both architectures. Therefore, the overhead for locating the data on a read miss in a COMA architecture is not expected to be significant.

A COMA protocol also must have a replacement strategy which makes sure that the last copy of an item is not lost when replacement occurs. One solution to this has been suggested by Hagersten et al. [6] where all shared copies are replaced with care. In order to guarantee some space for all items in a COMA, the address space in use cannot be larger than the sum of the attraction memories and the distribution of addresses evenly distributed over the sets in the attraction memories. Another solution has been proposed by Wallach and Dally [24], where each shared item has one tagged owner who replaces with care. Gupta et al. propose a strategy where each item has a defined home which is the synchronization point for the replacement action [3].

One important, and unique, property of COMA is its ability to dynamically adjust its ratio between replication and memory size according to the needs of the current applications [4]. Some applications have no need for massive sharing (replication), but need a large shared (physical) memory to avoid frequent disk accesses. Other applications, e.g., some database applications, benefit by large portions of their data being replicated among all the processors. In a NUMA architecture, the replication is limited by the size of its (second-level) caches, and its shared memory size is also fixed. The two application behaviors described above would need two different parameterizations of NUMA in order to run well. In a COMA, both behaviors could run well on the same architecture, thanks to its dynamic property.

A COMA will increase its replication until space runs out in the attraction memories. At this point, the amount of replication is determined by the size of the item space presently mapped by the operating system. A large, mapped item space results in a lower amount of replication, and vice versa, as shown in Figure 2. The operating system of the COMA can decrease the item space by reclaiming more pages, and increase the space again by mapping more pages.

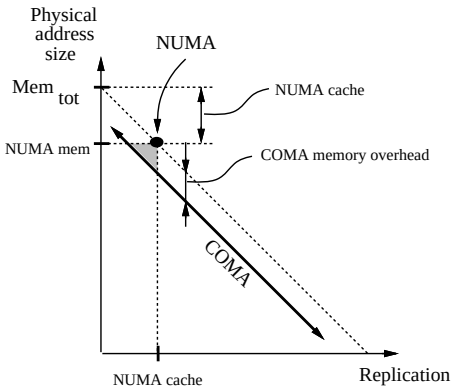

Figure 2: *A NUMA has a statically fixed relationship between the size of the caches (replication) and the physical memory, while a COMA dynamically can change its working point to suit the application.*

One drawback of the COMA architecture is the extra memory overhead required to make the local memory associative. This paper calculates this overhead to be between 1.5 to 6.5 percent of the local memory size. This memory overhead is marked as *COMA memory overhead* in Figure 2. In a NUMA, this memory could for example have been spent implementing yet another level cache, i.e. a remote access cache.

Figure 2 assumes a NUMA remote access cache larger than the *COMA memory overhead*, marked *NUMA cache*, and that the size of the first-level caches of the COMA and NUMA are neglectable.

3 Existing COMA node proposals

So far, we have explained why the dynamic behavior of attraction memories is preferable to the statically bound memories of NUMA architectures. However, if in implementing this dynamic property the hardware has a significantly longer access time for local accesses, the advantage of a COMA architecture over a NUMA is not clear. In this section we will review a few existing proposals for how the associativity of the attraction memories can be achieved. We study cache organizations which are direct-mapped, set-associative and an optimized version of set-associativity. They are compared to the organization used in the KSR1 architecture and the technique used in the DVSM proposal. All implementations are described with reference to a baseline architecture similar to the DDM prototype implementation [8].

3.1 Baseline architecture

The baseline architecture is shown in Figure 3. The *processor* is connected through its *first-level cache* to the local *data memory* (DM) on the local bus. Three additional units: BP, SM and AP, convert the data memory into the data part of an attraction memory. The *below protocol* (BP), snoops the traffic on the bus. The protocol makes lookups in the *state memory* (SM), and may abort (retry) any transaction due to non-matching address tags in the state memory and/or because the state of the matching data does not allow for the transaction. The below protocol also initiates necessary network transactions so that the requested operation on the local bus can be performed later. The below protocol may also provide parts of the address for the data memory based on which address tag in the state memory that matched the address on the local bus. The *above protocol* handles traffic from the network and (logically) shares the state memory with the below protocol.

3.2 A direct-mapped AM

In order for the *protocol handler* to determine if an item is stored in the attraction memory, each item is associated with an address tag in the *state memory*, which is compared to the most significant bits (MSB)

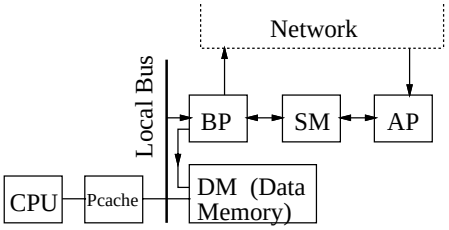

Figure 3: *The Base Architecture. (BP=below protocol, SM=state memory, AP=above protocol)*

of the requested address. A direct-mapped AM has each item mapped to a specific location, so there is no need to compare tags to determine which set an item resides in. For this reason, we can start the *read line* before the approval from the *below protocol* (tag comparison) is received. We rely on being able to restart a processor cache read before reading the last word of a cache line. In this case, the *below protocol* can wait until the very last cycle before deciding whether to force a retry or not. This allows for state lookup and data transfer to overlap. Only one access to the *state memory* is needed while several accesses to the *data memory* might be needed to transfer the whole cache line. Thus, *state memory* may use the same, (or possibly slower), memory technology as that used for the *data memory*,[2] and the delay in accessing the *state memory* will still be completely hidden, adding no extra latency caused by the functionality of the AM.

At first one might believe that the latency for accessing the *state memory* cannot be hidden on a write, since overwriting part of another item would be fatal. There is, however, full inclusion between a processor's (data) cache and its AM; in other words, there can be no copy of an item in the processor's cache unless there is also a copy of the item in the AM. This, together with the fact that a *write* to the attraction memory is never performed by the Pcache unless it already contains a copy of the item [18], can hide the *state memory* access—even from write accesses.

As can be seen in Figure 4, the implementation of a direct-mapped attraction memory is straightforward. The access time to data stored in *data memory* is equal to the *data memory* access time.

One major problem with this solution does however exist: Recent microprocessor designs can perform "critical word first" cache line fills, and/or complete the register file write back (from a load) as the appropriate word enters the cache. With these devices any retry for a cache line fetch should be issued before data is transferred. In this case then, the above proposal is not applicable.

[2]probably DRAMs

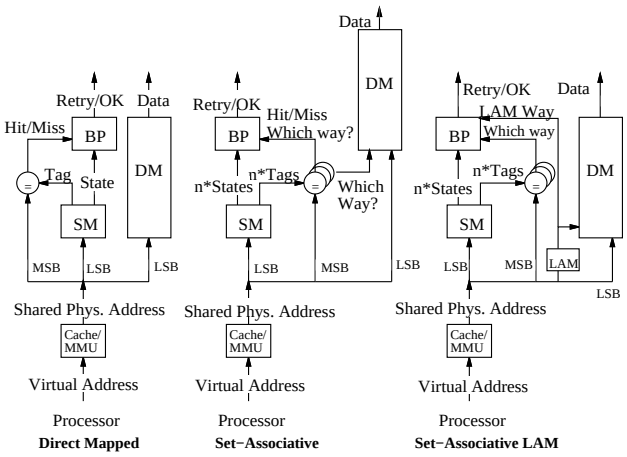

Figure 4: *Data dependency graphs for different ways of implementing associativity (BP=below protocol, SM=state memory, DM=data memory, and LAM=last accessed memory).*

3.3 Set-associative attraction memory

A direct-mapped cache (when possible) may be advantageous over a multi-way associative implementation for shortening access time to the AM, but it also increases conflict misses [9]. More associativity is expected to increase the hit rate in the AM. One can imagine situations for which a directly mapped attraction memory could be fatal for performance.

Another drawback of a directly mapped attraction memory is its limitation for replication of popular items. In order for one item to get replicated in all attraction memories, no other items for the same set of the attraction memory can be present in the machine; i.e., the item space cannot be larger than the size of one attraction memory. For two-way attraction memories, the item space can be half the sum of the AMs, and for four-way attraction memories, the item space can be three quarters of the sum of the AMs.

If the AM organization is multi-way set-associative, finding the location for the requested item is harder, since several possible locations exist for each item.[3] The address tag of all possible locations must be compared to the requested item's before the location in *data memory* can be determined. Small caches, implemented on a single chip, often access all possible data locations in parallel with the tag comparison, and select the right data at a late stage of the access. This

[3]Equal to the number of ways.

results in only a minor overhead compared to a direct-mapped implementation. Still, direct-mapped cache implementations have been justified for large cache sizes [9].

Accessing all possible data locations in parallel is complicated and impractical for the implementation of a large attraction memory with several ways, which is why the whole address tag lookup and comparison must be performed before the data access is started. This means putting the *state memory* lookup and the comparison on the critical path, as shown in Figure 4. As a result of its size, the *state memory* might be implemented with slow memory devices, resulting in a substantial overhead.

3.4 Last-access memory optimization

It is possible to implement a set-associative memory and retain the same low access latency for reads as described above for direct-mapped caches. A fast last-accessed memory (LAM) is added [4] to provide a guess as to which way in the set contains the item required. This memory contains one pointer of $\log_2(ways)$ bits per set, pointing to the way of the last accessed entry in the set. This is a most recently used algorithm - it assumes the way used in the set will be the same as the one used previously. As it requires only a few bits per set, the total LAM can be cheaply implemented in fast memory - therefore the cache item read can be started from the *data memory* using the way indicated by the LAM, and possibly aborted later if the tag check (from slower conventional memory) indicates the data is not present in the set or in another way. The LAM is updated if the initial guess turned out to be incorrect, and the same transaction is restarted after the abort, but with the correct LAM pointer.

If the *data memory* is built of DRAMs, the latency of the LAM can be totally hidden. The access to the LAM is made as the Row Address Strobe is applied to the DRAM. The LAM result is then ready to use as part of the Column Address.

A read access according to this scheme is shown in Figure 4.

The LAM optimization only works for read accesses. A write access is performed similarly to the set-associative implementation described earlier, since a write cannot be performed until the correct way has been determined.

The positive effects of returning the MRU data first in large multi-way caches have been studied by Chang et al. [2]. The LAM technique divides the AM into two parts, one with access time comparable to the *data memory* and one with a longer access time, similar to introducing yet another layer in the cache hierarchy. As such, the LAM strategy can potentially cut the access time for many applications, but is not expected to be successful for all applications. For a program that accesses a data set larger than the LAM part of the AM, the "LAM guess" may consistently turn out to be the wrong one.

As with the direct-mapped optimization above, we rely on being able to abort or retry the cache line read during the data transfer phase.

3.5 KSR1

It is hard to get full information about the KSR1 design and the facts presented here are partly based on assumptions and guesses. KSR1 has introduced proprietary solutions to many parts of its design [25, 13, 12]. This is also true for its caching system. Its first-level cache[4] appears to be large, 256 kbytes, but is organized in a somewhat unorthodox way. The cache is divided into associativity units of 2 kbytes. As a whole, the cache contains 128 such units, organized in a two-way set-associative manner, i.e., 2 x 64 x 2 kbytes. Each unit contains (among other things) one address tag, one "AM-way" pointer, and has space for 32 coherence units of 64 bytes of data and a few bits of state each. The replacement strategy is random.

The second-level cache (AM) is 32 Mbytes with associativity units of 16 kbytes, called a page, organized in 16 ways, i.e. 16 x 128 x 16 kbytes. Each page contains one address tag, some random information about the page's usage, and 128 coherence units of 128 bytes data plus some state bits, as can be seen in Figure 5.

On an access to a new associativity unit in the first-level cache, new space in the cache must be allocated (randomly). Secondly, the 16 possible locations in the AM are checked for a matching address tag. These 16 comparisons are (probably) performed partly sequentially in a scheme similar to the set-associative implementation just described. So, the overhead for bringing the first 64 bytes of an associative unit to the first-level cache from the AM is significant. The identity of the way that matches the address tag is stored in the first-level cache (AM-way), so that the next access to the same 2kbytes can be performed with no extra overhead. The data dependency of a KSR1 AM access when the AM-way information is available in the Pcache, can be found in Figure 6.

Allocating large associative units in the caches can

[4] Called "subcache" by KSR.

avoid the extra associative overhead for many accesses, as shown. It also cuts down on the memory required to store the address tags. The drawback of this scheme is a potentially low utilization of the data space in the caches. Even if only a single word is requested by the processor, 2 kbytes of the processor cache and 16 kbytes of the AM must be allocated, i.e., only 128 sparsely used words may reside in the data cache at the same time.

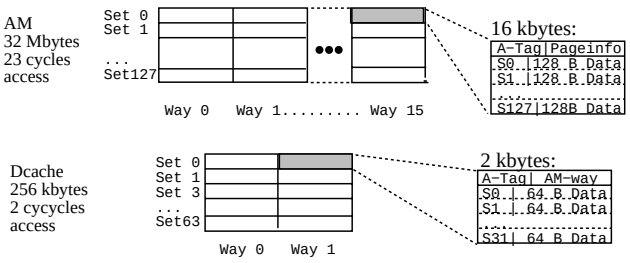

Figure 5: *The organization of the caches in the KSR1 architecture.*

There is no ordinary MMU functionality found in the KSR1. Virtual addresses are used as the global addresses in the system. This creates problems with aliasing and prevents efficient implementation of copy-on-write. The lack of page fault exceptions forces the search of a requested page in the whole machine before it can be determined whether or not a disk access is necessary. This should be compared to the early page-fault exception generated by an MMU. Furthermore it is not compatible with existing OS, compilers, and some applications, so a potentially large design effort is needed to rewrite portions of the software.

3.6 Distributed virtual shared memory

The title Distributed Virtual Shared Memory (DVSM) covers a range of multiprocessor shared-memory implementations where the coherence and migration of data between processors is maintained purely by software [15, 23]. DVSM systems utilize the processor's Memory Management Unit to detect and initiate coherence protocol actions, which are implemented in software. Just as hardware distributed shared memory systems, coherence traffic between nodes consists of messages on a network (for example, packets on an Ethernet). Most DVSM system implementations have COMA properties - the local (or main) memory of each processor is treated as a cache, with data items (pages) being allocated and invalidated, and data being moved and replicated from node to node without the notion of a fixed home which NUMAs have.

When a processor makes a "first-time" reference to a virtual memory address, the MMU has the responsibility to convert that virtual address into a physical memory address. In this case, the MMU will not have a translation in its Translation Look-aside Buffer (TLB), nor will an entry (physical-page pointer) be found in the current page table.[5] After failing to translate the virtual address, exception is taken, and the flow of the process is interrupted—a *page fault*.

In the event of a page fault, it is the responsibility of the operating system to allocate an unused physical page for the virtual page referenced. Further accesses by the application to the same virtual page "hit" in the TLB and are therefore completed without penalty.

The "first time" another processor access to data on the same virtual page, its MMU cannot perform a virtual access to physical address translation, so a page fault exception is generated. The operating system (or DVSM system) then allocates a new physical page in its memory. Data for the page is retrieved from the other processor node which already has a copy of the data corresponding to the virtual page being accessed. This is done by sending a request message to the other node, and then receiving the reply, which includes a copy of the data—much the same as one Attraction Memory requesting a copy of data from another in a COMA.

Aside from detecting the validity of an item (page), to maintain coherency we need to detect write accesses to a shared item (page)—for example when the item is shared across several nodes. To do this the page *write-protect* functionality of the processor MMU is used. By write-protecting a shared virtual page, read accesses proceed as normal, but write accesses cause the MMU to generate a *write-protect page fault*. Any coherency protocol of choice is implementable for DVSM.

Software (DVSM) COMA implementations have a number of advantages over hardware COMAs. The MMU functionality of the processor enables a virtual memory page to be mapped to **any** physical memory page on the local node. This enables a DVSM COMA to be built with a fully associative attraction memory, while hardware COMAs are restricted to either a direct mapping or limited associativity.

Being implemented in software means that DVSM systems can have more complex replacement and prefetching algorithms than would be reasonable to implement in hardware. The simplicity of a DVSM memory access can be studied in Figure 6.

[5] By either a software or hardware table lookup—depending whether the CPU's TLB is software or hardware loaded.

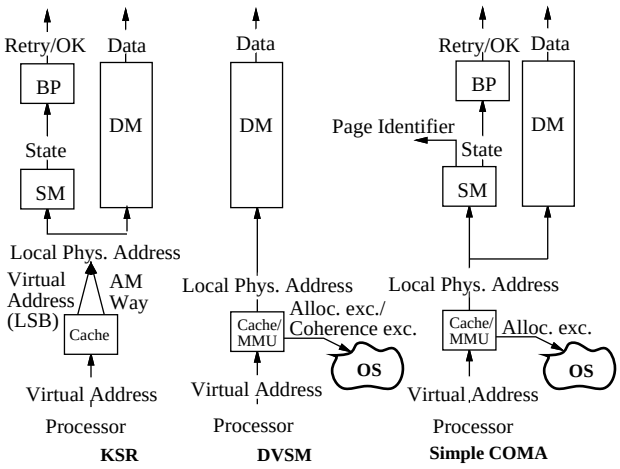

Figure 6: *The data access route for different attraction memory implementations.*

DVSM systems suffer from three problems. The first is the large item size—typically page sizes for today's microprocessors are 4 kbytes and growing. This large item size results in a potentially large amount of false sharing. For this reason more recent DVSM systems are turning to weaker memory consistency models to achieve performance.

The second problem is the long latency associated with a cache miss. This, perhaps surprisingly, is **not** due to the cost of taking a page fault—processors with software loaded TLBs [17] illustrate that page faults can be dispatched and dealt with in only a few tens of cycles. The costs are associated with the creation and dispatch of messages on more traditional networks such as Ethernet.

Finally, and most importantly, the processor must *also* deal with the coherency traffic from other nodes as well as its own. Hardware COMAs such as the DDM and KSR1 can deal with coherency traffic from other nodes on the network without disturbing the local processor.

4 The simple COMA

We have seen in earlier sections how a COMA can be built on an arbitrary communication network, and how the coherence protocol of a COMA is very similar to NUMA coherency protocols. So where is the complexity in a COMA ?

From the DDM and the KSR1 it appears that attraction memory implementation are complex. This would not be an accurate conclusion, as both the DDM and KSR1 are early all-hardware implementations of COMAs. We believe the right implementation solution for a COMA results from combining the best features of the complex all-hardware COMA approach, and the simple-but-poor-performance software DVSM approach.

4.1 A better COMA implementation

Here we describe a solution to the problem of attraction-memory implementation, which we believe is simpler and faster than the previously described solutions. Our proposed implementation has a fully associative attraction memory *with* a short access time. It is far simpler than the KSR1 implementation, and eliminates some of the disadvantages found in the KSR1.

COMAs differ from DVSMs mainly by their smaller coherence units (items), and by the coherency protocol being implemented in hardware, rather than in software. We propose to retain some of the DVSM software functionality, while additional hardware support, similar to the DDM implementation, is added to decrease the size of the coherence units and also make the coherency protocol implementation more efficient.

Our proposal is a combination of fully associative mapping using the MMU, as seen in DVSM, in combination with a coherence protocol implementation similar to the protocol handler of the baseline architecture.

4.2 The proposed COMA node

Ignoring issues of network and protocol as orthogonal, we propose a COMA node designed as follows. Just as with DVSM implementations, the allocation and replacement of items within the attraction memory is handled by software — performed necessarily at page-size granularity. This enables our COMA system to have a fully associative attraction memory. Unlike DVSM, coherence actions will not cause MMU exceptions, the addition of state memory and a simple protocol handler (PH) enables coherence checks to be performed on a per-item granularity - typically a first- or second-level cache line size.

Therefore the *state memory* holds protocol state bits per *attraction memory* item. Unlike the DDM implementation, however, there is no address tag stored with each item. We do not need to validate access to an item with an address tag, since we performed the item identification validation effectively with the MMU.

An additional state memory (or part of the *state memory*) holds one *page identifier* for each page (of items) in the *attraction memory*. The page identifier

(PI) is assigned by the software which allocates the virtual-to-physical attraction memory page mapping. This page identifier is used by the protocol handler to identify a shared page when communicating with another node. In order to uniquely identify a page , a 20 bit identifier is enough for a machine with up to 4 Gbytes held in 4 kbytes pages.

Note that the number of bits for the page identifier in no way affects the virtual addressing capabilities of the processors, just the total number of physical memory pages in the machine. Figure 7 compares the *state memory* in the proposed solution to that in the DDM.

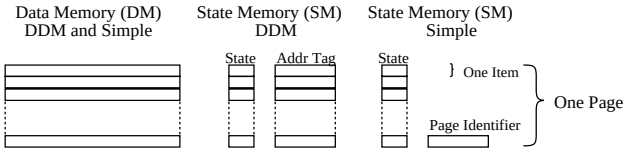

Figure 7: *Comparison of the DDM and Simple-COMA state memories*

Figure 6 illustrates the memory access paths of the proposed simple COMA alongside those of a basic DVSM system, and the KSR1 architecture. Access to the attraction memory may be started simultaneously with the state memory lookup. This is because the MMU performs the associative cache lookup for the item position. When the physical memory address appears, for both the *state memory* and *data memory* lookup, there is a **direct** mapping for an item within the selected page. This functionality enables the state memory to be implemented using the same speed devices as the attraction memory - for example conventional DRAM. The MMU has already tested the validity of the mapping (item identification) in the attraction memory, and so we can start either a read or a write knowing that the item either exists or will exist at that location. A miss caused by an unfavorable item state can simply abort the operation as part of the coherence actions taken.

4.3 Implementing remote associativity

So far we have described the mechanism by which local processor memory accesses are directed to the correct data item, and access is validated by the *state memory*. We have not discussed how the coherency protocol handler on one node can communicate with its counterpart on another processor node.

When a local memory access fails, for example because of a read to an invalid data item in the *attraction memory*, the page identifier is produced by the smaller *state memory*, or possibly even from a table in main memory. This PI is used in a message to another node to retrieve a copy or exclusive ownership of the missing item.

When the protocol message arrives at the destination node, that node must, from the page identifier, be able to lookup **its local** physical page mapping in order to find its copy of the item and state. There are several methods of performing this reverse PI to physical page translation.

Page identifier CAM
A PI CAM is a Content Addressable Memory - when an incoming PI is applied, the PI CAM can return the physical page number corresponding to that page on the local node. When a physical page is allocated on a node, the entry in that node's PI CAM allocated for the physical page chosen is filled with the PI given to the page.

For a processor node with say 64 Mbytes of RAM organized in 4 kbytes pages, a PI CAM with 16384 entries is required, something which is practicable with today's technology.

Page identifier MMU
A simpler solution to the fully associative memory as described above, a kind-of MMU functionality, can be implemented. A small associative cache (like a TLB) holds the most recent PI conversions, and a PI table is walked when an entry is not found in the PI-TLB.

It is not clear how effective this will be in practice compared to the PI CAM, since the PI MMU receives coherency traffic from potentially all other nodes in the system. Such traffic will not have as much locality as the memory accesses from the single local processor (the latter fact exploited by the processor's own TLB). The PI-TLB may have to be several times larger than one might allocate for the CPU.

A similar functionality to this is already implemented for the "DMA Engine" (Elan chips) in the Meiko CS-2 network communication interface [16]. This is used under direct processor control to transfer messages from one processor's memory to another's.

Physical pointers
This scheme effectively moves the hardware complexity of a PI-MMU or PI-CAM into software. In addition it reduces the latency of a remote access to data, at some extra cost to the page replacement and the protocol implementation.

The idea of this scheme is that the page identifier should be the physical page number of the corresponding page on another node (and possibly also some form of node identifier). For example if node A shares a page copy with the owner of the page — node O, then

node A will have the physical page number of the page on node O as its page identifier.

When node A wishes to perform some protocol action (such as item invalidation), it sends a message to node O. As the physical page number is given with the message access to the correct state memory slot can be started immediately. There is no need for the delay associated with the PI-CAM or PI-MMU.

To complete a protocol implementation, there must be some way of identifying the physical pages allocated on the sharing nodes (node A in the example above). This is achieved by storing their respective physical page numbers with the item copy set information.

As with other hardware COMAs or NUMAs, the copy set (and reverse page identifiers) may be held in any form: as linked lists such as SCI [10], or even in hierarchical directories as in the DDM.

Note, we introduced the notion of an "owner" node merely as a focal point for identifying the copy list. Since pointers are allocated and reclaimed by software, ownership of a page may be easily made to move.

While providing faster access when a coherence message arrives at a node, this scheme requires extremely careful pointer allocation and reclamation. A physical page may not be reallocated until all other nodes which hold pointers (node and physical page number) to it have this mapping invalidated. This may require expensive interprocessor interrupts and synchronizations. All said and done, the one or two cycle extra cost of the PI-CAM or PI-MMU may be of no significance in the face of a typical 100 cycle network latency.

Experiments will indicate which of the above three schemes is likely to be the most effective. An attractive solution might, once more, be not to choose either the simple (MMU) solution or the efficient (physical pointer) solution, but rather a combination of all three solutions and thus combining simplicity and efficiency.

4.4 Potential drawbacks

Using the MMU to perform the attraction memory item lookup has one potential drawback. It is possible for an application to access only one item on each memory page. This commits the other unused items on the same page to particular virtual memory addresses; the physical memory cannot be used for other virtual items.[6].

[6]This is not quite true. With care, two virtual pages can be mapped onto the same physical page, provided it can be guaranteed that the accessed items in each of the respective virtual pages do not overlap in the physical page. This requires some knowledge of the application by the OS.

In the worst case, a machine with 64 Mbytes of memory could share only 4096 items (if using a 4 kbytes page size). With an item size of, say, 128 bits (16 bytes), this enables a maximum shared data space of 64 kbytes before thrashing starts taking place — such thrashing involves full page faults. This might be a potential problem and more statistics are needed before we understand its impact. Simple COMA is better than the KSR1 though, which uses a similar allocation scheme with larger allocation blocks (16 kbytes).

5 Performance and complexity

When comparing different implementation proposals it is important to deal with both the performance and the implementation complexity. Here we will qualitatively discuss the performance and complexity of the direct mapped, the set associative, the KSR and the simple approaches to building a COMA.

5.1 Performance

The performance of a COMA implementation is characterized by the hit rate in the attraction memory and first-level caches, and by the latency for the different types of accesses. This study focuses on the node implementation techniques and does not cover in detail the differences in latency for remote accesses.

Typical applications generally have good hit rates for attraction memory accesses [5]. To achieve good performance in a COMA machine it is essential to minimize the latency for these hits. In the case of an attraction memory miss, the latency differences due to these node implementation considerations are not so significant since they are part of a much larger latency associated with the remote access. Other implementations such as network structure and COMA directory policies are, of course, also highly important but lie outside the scope of this study.

The Table in Figure 8 summarizes the main characteristic differences for the implementation alternatives considered in this study.

Note that direct mapped has a lower hit rate for the same size of attraction memory since it might suffer from an increased number of conflict misses. It also has limitations on the degree of replication that can be utilized. We also assume that there is full inclusion between the primary cache and the attraction memory.

The KSR can be expected to have a lower hit rate in its Pcache since it has very large allocation blocks and will suffer if the accesses are sparse and spread in the

Impl.	*Access time*		*Comment*
	Pcache	AM	
Direct mapped	fast	fast (DM access)	Lower AM hit rate.
Set assoc.	fast	slow (SM+DM acc.)	
KSR	fast	fast or slow (DM or SM+CMP+DM)	Proprietary Pcache with lower hit rate.
Simple	fast	fast or slow (DM or TLBfill+DM)	Fully associative AM.

Figure 8: Main performance characteristics for the different implementation strategies.

address space. On the other hand it is fair to assume that the caches can be made larger using KSR's technique since the associative part is comparably small.

The fast access times for the KSR AM applies for blocks that have already been allocated in the primary cache. For the first access after a block has been replaced from the Pcache, the longer access time applies.

The Simple and the KSR alternatives might suffer from reduced AM hit rate since they both have large allocation units. If accesses are sparse the effective size of the AM can be significantly smaller than it nominally appears to be.

At a first glance, it looks like the proposed solution is not that different from a NUMA where pages can migrate, which has been proven useful [22]. The similarity is true for read-only data. However, simple COMA can also allow replication of read-write data and still maintains coherence between the many copies. Also, different parts of a page may migrate to different nodes in the simple COMA, at the expense of wasted memory, suiting a wider range of applications. The migration decision is also made less fragile. Instead of using sophisticated hardware solutions to figure out where the only precious copy of a page should be kept [22], a more liberal strategy can be used. Every node that needs a copy of the page as a whole, or parts there off, is given a copy. A page reclaiming algorithm in each node decides about when a page is no longer in use in the node, and therefore should be reclaimed.

5.2 Complexity

All the hardware-based implementation proposals differ from the DVSM approach in that hardware complexity is added to improve the performance of the system. The central problem is of course how to achieve the highest performance with the least hardware. While this is normally hard to do, it is desirable to find solutions that with a limited hardware complexity give a high performance.

We have studied the hardware needed to implement each scheme. We assume a machine of 500 nodes each with an attraction memory of 64 Mbytes. We assume a word size of 64 bits. The size of a page is 4 kbytes and the coherence unit is 4 words (32 bytes).

Note that we do neither consider the hardware needed for the interconnection network nor the directory information needed to find remote copies in the machine. These are properties that are orthogonal to the node implementation which is the focus of this paper.

Direct mapped
The direct mapped approach is probably the most straightforward method to implement a COMA. Each node needs a *state memory* that contains state and tag information for all 4-word items in the node. 64 Mbytes of data makes 2 Mitems. The total memory space in the machine is $64\ Mbytes * 500 = 32\ Gbytes$. To address this the item identifiers need to be 35 bits. The lower 26 bits are needed to address within the attraction memory. This leaves 9 bits for tags associated with each item entry. If we reserve 4 bits for state information the net result is 13 bits per item ($32 * 8\ bits = 256\ bits$). This leaves us with an overhead of $13/256 = 5\ percent$.

Set associative
If we assume a 16-ways associativity we get the following overhead calculations:

The 64 Mbytes attraction memory contains $2\ Mitems/16 = 128\ ksets$. These are addressed with $17 + 5 = 22\ bits$, leaving 13 bits for tag. The memory overhead evaluates to: $(13 + 4)/256 = 6.6\ percent$.

In addition to the memory overhead we also need hardware for the associative comparison. In this case sixteen 13-bit comparators are needed for the attraction memory. The state memory also requires to be able to support all the $16 * 13 = 208$ bits simultaneously. The alternative is to make the comparison in serial. This requires less hardware but further increases the access latency.

KSR
Unlike the other proposals, the KSR approach requires

special hardware at the primary cache level. This rules out the use of off-the-shelf processors with on-chip caches. Although the actual overhead is very small (0.2 percent) the inconvenience of a proprietary design might be substantial.

The overhead of the attraction memory state and tag is reduced in the KSR approach since the AM is divided into 16 kbytes pages. Tag information is only stored with each page while state still must be supplied with every item. Similarly to the set-associative case, the tag associated with a page is 13 bits. This is however negligible compared to the 16 Kbytes data of that page. The state overhead is $4/256 = 1.5$ *percent*.

Simple
The Simple approach uses the standard MMU for associativity. State information still has to be supplied with each item as in the KSR case. The overhead is also 1.5 percent.

Complexity summary
The complexity properties for the proposals are summarized in the following table:

Impl.	*AM Overhead*			*Off-the-shelf-comp*
	Tag	State	Other	
Dir. map.	3.5%	1.5%		YES
Set assoc.	5.1%	1.5%	Wide mem cmp	YES
KSR	0%	1.5%		NO
Simple	0%	1.5%		YES

Figure 9: Complexity summary.

As can be seen, both the KSR and the simple proposals have very small overhead, while the simple proposal can be implemented with standard hardware and also gives a fully associative attraction memory.

6 Related work

6.1 Wind tunnel

In an attempt to create an efficient parallel simulator on the CM-5 for simulation of shared memory architectures, a research group at the University of Wisconsin came up with a solution similar to the one described here [20]. That solution combines the DVSM coherence mechanism with support for coherence units smaller than a page. Both allocation exceptions and coherence exceptions are handled by software, similarly to the DVSM, but validation checks of access right can be made with a finer granularity than a page.

A bogus ECC code is simply set for the memory of all "invalidated" cache lines of a page. Only accesses to these cache lines will generate exceptions. A problem arises if cache-lines with different access priority co-exist on a page, sine the ECC code cannot differentiate between read and write accesses. Instead, the whole page will be write-protected, causing some valid writes to cache lines to create exceptions, a potential drawback of this scheme. The advantage is that it makes use of an existing architecture and its ECC implementation.

6.2 The Sun S3.MP project

The S3 MP project [19] aims to build a distributed shared memory architecture by adding a relatively small amount of electronics to an existing workstation's memory bus (typically a SPARC-10).

The S3 operates by partitioning the main memory of the local processor into a local main memory partition, to be used as usual, and a cache partition for remote memory accesses. This cache size is programmable — a distinct advantage over conventional NUMAs which have their cache size fixed by hardware; however, it does not allow a dynamic trade between replication and problem size as COMAs do, since the cache size is determined when the machine is booted.

The S3 MP includes hardware to convert local memory addresses into a "global 64 bit address." Similarly, hardware converts such global addresses from the network into local memory addresses.

The design philosophy has much in common with the simple COMA we propose: use existing workstation technology, simple coherence, and communication hardware as an extension to the main memory operating as the cache. However the result of the S3 MP is still a cache-coherent NUMA; there is still the fixed cache size; and fixed home.

7 Conclusion

A COMA node has been believed to be slower and more complex to implement than alternatives. In this paper we propose a simple and efficient COMA implementation based on existing commercially available components.

In spite of its new structure and behavior, the implementation presented conforms to existing assumptions about shared memory architectures made by op-

erating systems, compilers, and applications, and actually adapts to their behavior.

Acknowledgements

Several useful comments on an earlier draft of this paper was provided by Rafael Saavedra. SICS is sponsored by Asea Brown Boveri AB, Ericsson AB, IBM Svenska AB, Televerket (Swedish Telecom), Försvarets Materielverk FMV (Defense Material Administration), and the Swedish National Board for Industrial and Technical Development (Nutek).

References

[1] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 4th Annual Architectural Support for Programming Languages and Operating Systems*, 1991.

[2] J.H. Chang, H. Chao, and K. So. Cache Design of A Sub-Micron CMOS System/370. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 208–213, 1987.

[3] A. Gupta, T. Joe, and P. Stenström. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. TR #CSL-TR-92-524 Stanford University, 1992.

[4] E. Hagersten. *Toward Scalable Cache Only Memory Architectures.* PhD thesis, Royal Institute of Technology, Stockholm/ Swedish Institute of Computer Science, 1992.

[5] E. Hagersten, M. Grindal, A. Landin, A. Saulsbury, B. Werner, and S. Haridi. Simulating the Data Diffusion Machine. In *Proceedings of Parallel Architecture and Languages Europe.* Springer-Verlag, 1993.

[6] E. Hagersten, S. Haridi, and D.H.D. Warren. The Cache-Coherence Protocol of the Data Diffusion Machine. In M. Dubois and S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors.* Kluwer Academic Publisher, Norwell, Mass, 1990.

[7] E. Hagersten and A. Landin. An Initial Attempt to a General Network COMA. DDM-memo, SICS, August 1991.

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

[9] M. Hill and A.J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[10] D. James, A.T. Laundrie, S. Gjessing, and G.S. Sohi. Scalable Coherence Interface. *IEEE Computer*, 23(6):74–77, June 1990.

[11] T. Joe, J. P. Singh, A. Gupta, and J Hennessy. An Empirical Comparison of the Kendall Square Research KSR1 and the Stanford DASH Multiprocessor. Presented at the Third Workshop on Scalable Shared Memory Multiprocessors (in connection with ISCA), May 1993.

[12] U.S. patent 5,005,999 Multiprocessor Digital Data Processing System, October 1991. Kendall Square Research.

[13] Technical Summary, 1992. Kendall Square Research.

[14] 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.

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

[16] Meiko Ltd. CS-2 Product Description, 1992.

[17] MIPS. R4000 Users Reference Manual, 1992.

[18] Motorola. *MC88200–Cache/Memory Management Unit, User's Manual.* Prentice Hall, New Jersey, 1989.

[19] A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Browne, G. Aybay, and D. Lee. S3.mp: A multiprocessor in a matchbox. Technical Report parcftp.xerox.com:/pub/dlee/PASA_proc.ps, Sun Microsystems Computer Corporation, 1993.

[20] 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.

[21] P. Stenström. A Survey of Cache Coherence for Multiprocessors. *IEEE Computer*, 23(6), June 1990.

[22] P. Stenström, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, 1992.

[23] T. Stiemerling, A. Saulsbury, and T. Wilkinson. A DVSM server for Meshix. In *Symposium on Experiences with Distributed and Multiprocessor Systems III*, March 1992.

[24] D. Wallach. A Scalable Hierarchical Cache Coherence Protocol. SB Thesis. MIT AI lab, May 1990.

[25] D. Windheiser, E. L. Boyd, E. Hao, S. C. Abraham, and E. S. Davison. Analysis of Latency Hiding Techniqueus in a Sparse Solver. In *Proceedings of IPPS*, 1993.