Advantages of COMA

John G. Robinson

David C. Baxter

Jim Gray (U.C. Berkeley)

Kendall Square Research

July 6, 1995

Table of Contents

1 Executive Summary

Demands for high performance computing in both the commercial and scientific markets have led to multiprocessor computers with distributed memory. This reports summarizes the development of these machines and presents case studies that illustrate the benefits of implementations based on a ***C***ache-***O***nly ***M***emory ***A***rchitecture (COMA).

The compelling features of COMA are:

- Node (DRAM) memory is dynamically allocated between local memory and cache, based on the access patterns of the applications that are running.
- Data migrates to where it is being used, without extra software intervention.
- Capacity misses from the node are reduced to near zero, since nearly all of the node's DRAM memory can act as interconnect cache, as needed. The benefits of this are reduced average memory access time, and much less performance pressure on the interconnect.
- The relative ease of programming with shared memory instead of message passing.

COMA is superior to other distributed-memory multiprocessor designs. The best possible performance requires elimination of all remote memory accesses other than those caused by data initialization or updating of data shared among separate nodes. COMA can achieve this optimum. This article details three case studies in which COMA needs two or three times fewer remote memory accesses than other designs. As processors get faster and memory latency becomes even more critical, the performance advantages of COMA become even greater. COMA architectures are the preferred design for scalable computers.

COMA latency and bandwidth characteristics enable scalable computers to be built from commodity processing nodes. Each node can be a few microprocessors on a shared bus -- typical of the Intel P6 products appearing today. For plausible designs using today's commodity microprocessor systems and a small (~10%) incremental hardware investment, COMA allows construction of a multiprocessor system that scales from 4 to 32 processors with a predicted scaleup of 75% for OLTP applications, which is greater than can be expected from competing architectures, such as ***C***ache ***C***oherent ***N***on-***U***niform ***M***emory ***A***ccess (CC-NUMA). (See Figure 1.) This gives an 8-node cluster of 4 processor nodes an effective power of 24 times that of a single processor (See Figure 2 for two configurations based on KSR AllCache (R) technology and commodity PCs). Similar scaling and performance can be expected for decision support system (DSS) applications, as shown in the case studies of this paper.

COMA performs better for a given product investment than the best alternative architecture, CC-NUMA, with as much as a 2:1 advantage in remote memory accesses, and an overall speed advantage of 4:3 or greater. A COMA machine is effectively similar to a CC-NUMA machine with a large enough interconnect cache to eliminate nonessential remote memory accesses. The performance benefits, due to properties of shared memory applications, will grow with improvements in scalable applications, which will in turn enable even larger COMA machines to be built with similar performance benefits.

COMA is applicable to a wide range of system design approaches. Issues such as the granularity of inter-processor memory transfers and the latency and bandwidth of the inter-processor interconnect are largely independent of the choice to use COMA, although they are important considerations in designing or evaluating any distributed shared memory (DSM) machine. COMA can be implemented using point-to-point or broadcast interconnect topologies, and finding the data can be accomplished via a fixed partitioning of the directories (COMA-Flat) or via a search. COMA does bias some of these trade-offs. For example, because capacity misses are reduced to near zero with COMA, the interconnect can be optimized for handling coherency misses, which makes broadcast topologies interesting.

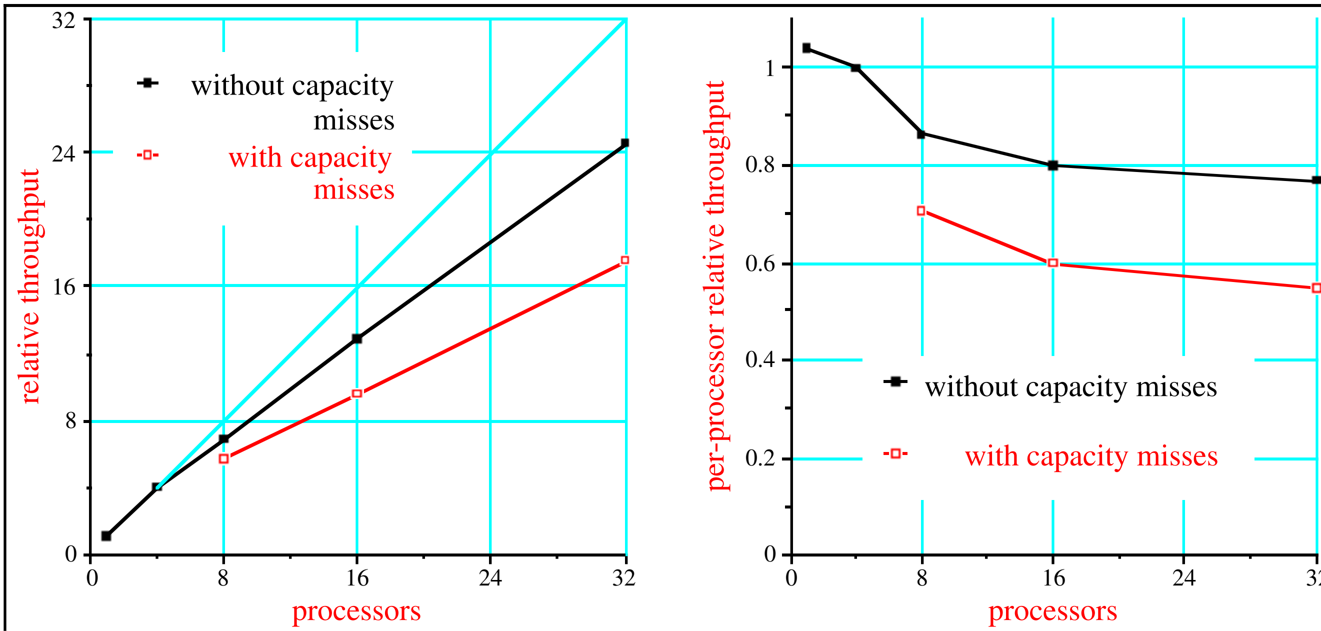

Figure 1. Modeled On Line Transaction Processing throughputs (TPC-B-like transactions) for 1, 4, 8, 16, and 32 processor systems. The model is based on Oracle7 measurements on a KSR-1. The left graph shows the relative throughput. As the number of processors and disks increases, the throughput rises nearly linearly -- 32 processors have the power of 24. The right graph shows the per-processor relative throughput -- at 32 processors, each processor is delivering about 78% of its power. The lower curve in each graph shows the performance cost of capacity misses, assuming a CC-NUMA with an interconnect cache that suffers equal numbers of coherence and capacity misses, as defined in section 2.1. A COMA machine need not suffer any capacity misses. More details may be found in this paper and in [Hi95].

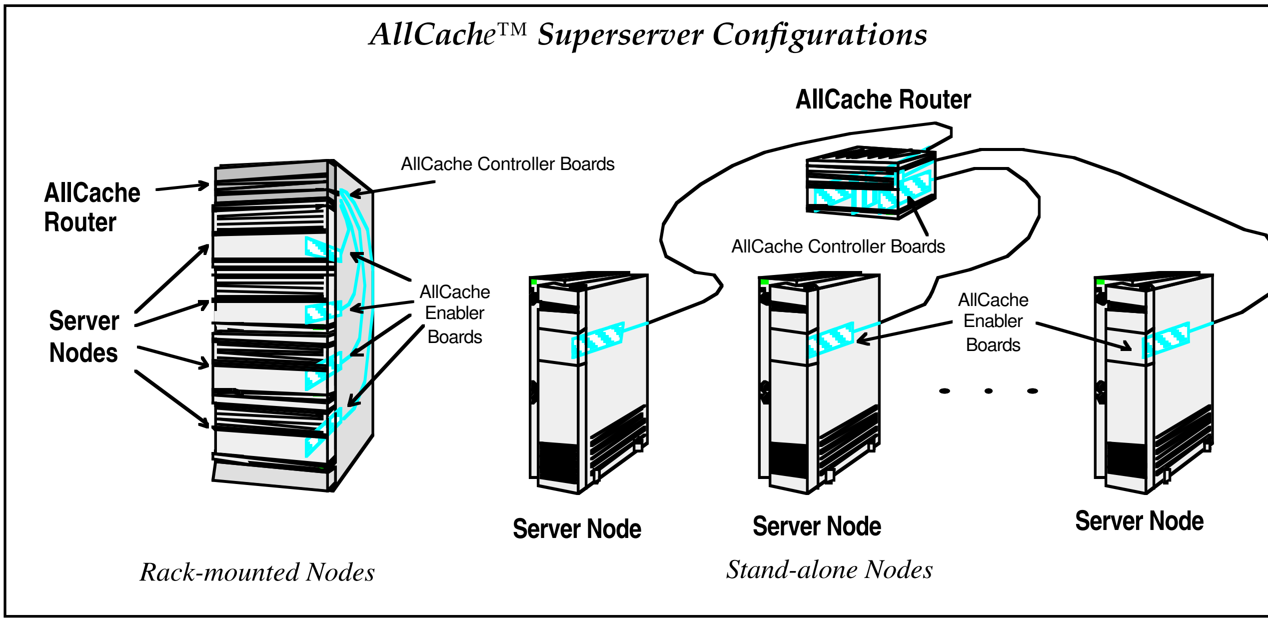

Figure 2. Two possible AllCache Superserver COMA Configurations. One can build a rack-mounted specialized system or a stand-alone router system. Both designs use standard microprocessor-based nodes (e.g. Intel P6 with PCI.) They add an AllCache Enabler board on the MESI memory bus to integrate the memory of each node into the global AllCache memory. The AllCache Enabler board fits into a standard slot on the system board, providing direct access to the memory bus. This hardware and KSR software layered on the host operating system are the only non-standard components.

2 Multiprocessor Classifications

Multiprocessors executing independent programs are often called ***M**ultiple **I**nstruction stream, **M**ultiple **D**ata stream* systems (MIMD). Most multiprocessors are MIMDs, including KSR's first two machines[Ke92]. Although each processor has its own instruction stream, the processors together may be trying to solve a single problem. This ability to cooperate allows multiprocessors to solve large scientific and commercial problems at lower cost than traditional supercomputers or mainframes. The advantages of multiprocessors for scientific computations are well documented. This paper shows that the same benefits accrue to commercial applications such as On-Line Transaction Processing (OLTP) and Decision Support Systems (DSS).

Multiprocessors are not a new idea. They typically arise to address two needs. At the highest end of the performance scale, connecting two or more of the fastest processors available yields an even faster machine capable of greater throughput and of solving even larger problems. Virtually all super-computers since the Cray 1 have been multiprocessors. At the other end of the scale, a multiprocessor architecture can yield a cost-effective machine by taking advantage of the superior price/performance ratio of commodity processors produced in high volume, compared to the fast but exotic and low volume processors in mainframes or super-computers. This trend began with the minicomputers in the 1970s, and made a transition to microprocessors in the 1980s.

The industry generally agrees that a shared-memory programming model is preferred and most programs are written with a shared memory model in mind. Conventional wisdom until recently has been that shared-memory multiprocessors cannot scale beyond a dozen processors. We focus on new approaches to shared-memory machines that scale well beyond this limit. These architectures are well suited to the coming generation of commodity microprocessors from Intel and others.

2.1 Memory Hierarchies

One may view the history of computer architectures as largely the history of memory hierarchies. The foundation of modern designs dates back to the Atlas project at the University of Manchester [Ki62], which proposed that main memory be used to store only the most frequently needed portions of the address space of a program, with the balance being saved on a disk until needed. This freed the programmer from worrying much about the physical limits of the computer's main memory and instead created the illusion that a much larger ***virtual*** memory was available. All modern general-purpose systems support this abstraction.

As processor speeds outstripped memory speeds, a similar technique developed to bring the most frequently-needed portions of the main memory into a smaller faster ***cache memory*** that the processor could reach more quickly. Cache memory has become a ubiquitous component of general-purpose computers. Now most systems have at least a three-level memory hierarchy: cache, main and disk. An even smaller and faster on-chip cache is now included in many processors, even in the inexpensive PC class, creating a four-level hierarchy.

The goal of the memory hierarchy is to provide the program the illusion of a very large, fast memory without needing to buy all of the very-fast memory components to implement it. Spatial and temporal locality of reference enable statistical techniques to manage the contents of memory at each level.

Another trend allows multiple processors and their caches to share a single memory. Since the multiple caches may each have copies of a given location in memory, they must communicate with each other to present their processors with a consistent view of the data. When the caches and main memory share a single bus, the typical approach uses a family of protocols that support a technique called ***snooping***. This allows a modest number of processors to share data, with the illusion that they all see the same memory contents at the same time (at least, when program correctness requires it), yet with almost all the performance gain afforded by caching.

Each processor operates at full speed when the data needed for its operation are in the cache. When data are not present in cache, the processor suffers a ***cache miss*** and performance slows. The cache miss rate is a critical component of system performance. To amortize the cost of a cache miss and simplify cache management, data are brought into (and expelled from) the cache in units larger than one memory word, called cache ***lines***.

We identify three types of cache misses. The first time a processor references a memory location in a given cache line, it must be brought into the cache, due to an ***initial*** cache miss. As cache lines are used, the cache eventually fills with useful data. Once the cache fills, cache lines must be replaced, or ***evicted***, to make room for new data. When previously cached, but evicted, data are referenced again, the consequent cache miss is referred to as a ***capacity*** miss. Enlarging the cache lowers the frequency of capacity misses. As a rule of thumb, a cache must be made eight times larger to lower cache miss rates by a factor of two under general-purpose workloads [Ma94].

In a multiprocessor system, a third type of cache miss, known as a ***coherence*** miss, occurs when one processor modifies a cache line that is shared. This can only be done safely if all copies of the line in other processors' caches are removed (invalidated). Coherence miss rates are a property of the program’s algorithms and cannot be lowered by application of more or smarter hardware.

2.2 NORMA distributed memory multiprocessor

The simplest multiprocessor design has ***NO R***emote ***M***emory ***A***ccess. All memory is local to one node, and all inter-node communication is via low-overhead interprocessor messages. These architectures are often referred to as “private memory”, “shared nothing”, “message passing” or “multi-computers”. A further sub-classification designates some of these machines as shared-nothing, where no I/O devices are shared between processing nodes, and shared-disk, where two (or more) ports of the disk controllers are attached to separate nodes. Much has been written about these two models for database and transaction-processing applications [St86]. Many multiprocessors have been built with this model, including some with thousands of processors.

NORMAs are the cheapest to build, as one may employ existing single- or multi-processor systems as nodes in the machine, and interconnect them with specialized I/O devices, or general-purpose packet networks. But NORMAs are hard to program. Achieving scalability and good performance requires a good mapping of the problem at hand into the multiple computers of the machine. If one can map to this model, the resulting application will have good price-performance. NORMAs also have natural fault isolation and so have good fault-tolerance.

Examples of NORMAs include the IBM Sysplex and SP2, DEC VAXcluster, Intel Paragon, nCUBE, Tandem, Teradata, and Thinking Machines CM-5.

2.3 UMA central memory multiprocessor

In some architectures, multiple processors share a uniform centralized memory. All accesses to this shared memory take the same length of time, so these machines are known as UMA, or ***U***niform ***M***emory ***A***ccess time machines. Although they may be built without any caches or memory coherence, we focus on machines that include a cache for each processor, and use a snooping bus protocol for coherence.

UMA software may assume coherent shared memory, and need take no special action to guarantee that every processor sees the same data at the same location at a given point in time. UMA performance limits arise from saturating the shared bus connecting the processors to the memory. For 386-class processors, as in the Sequent machine, the performance limit is reached at or before 32 processors, and is reached at 16 or fewer Pentium class processors. Processor speeds have increased faster than bus and memory speeds and latencies. Newer processor generations reach bus saturation even sooner. This progression is inevitable, since bus speed limits arise in part from physical limits such as the speed of light. By reducing capacity misses, larger caches can allow more processors to share the bus, but even this has its limits. From a price-performance standpoint, this approach is self-limiting. The cache cost per processor grows faster than the number of processors the bus can support.

In some of the architectures we describe, the processing units of the machine will themselves be UMA multiprocessors, typically with relatively few processors. We use the term ***node*** with this connotation.

Examples of UMA architectures include the Compaq ProLiant 4000, IBM 3090, DEC VAX and Alpha, Encore, Sequent, Silicon Graphics and Data General Aviion machines. The Cray XMP and YMP super-computers are examples of UMA machines without caches.

2.4 NUMA distributed memory multiprocessor

UMA machines evolved to distributed shared memory computers with ***N***on-***U***niform ***M***emory ***A***ccess times to local and remote memory. NUMAs have a single shared memory implemented by interconnecting a number of nodes' memories with a high-speed interconnect. Some of the shared memory is local to each node. Remote memory is accessed in normal memory cycles, but the access times are typically an order of magnitude greater than local memory. NUMA shared memory size scales with the size of the machine -- an important advantage compared to bus-based machines. The interconnect cost typically grows faster than the machine size, but may not become significant before reaching very large node counts.

NUMA application software attempts to reduce the frequency of remote accesses by assigning each virtual page to a physical page in the node where it will be accessed most frequently. As with the NORMA machines, the performance of NUMA application software depends on assigning data to local memory on the node that wants to use it.

A further limitation of the original cached NUMA machines was that the remote memory was not cache-coherent with the local processor's caches (the first BBN Butterfly machines had no caches). As caches became necessary for processor performance, the problem of cache coherence arose, since there was no way for one processor's caches to know what the processor caches in other nodes were doing. Software must maintain cache coherence by making shared data uncacheable (which gives up the performance gain of caching), or by flushing or invalidating cached shared data at appropriate times and using locks. Hence software for these machines is quite specialized both for correctness and performance.

Examples of this architecture are the Cray Research T3D[Cr93] and BBN TC2000.

2.5 CC-NUMA

The cache coherence limitations of the original NUMA machines led to the addition of hardware to maintain ***C***ache ***C***oherence among the nodes, These machines received the acronym CC-NUMA. CC-NUMAs are under active development at University laboratories and in industry.

CC-NUMAs maintain coherence by assigning each shared-memory cache-line a ***home node***. Cache ***directories*** at each node track the current state of each line at that node. Each node holds a portion of the global directory, so that the directory itself doesn't become a hot spot.

Typical CC-NUMA designs use cache misses to trigger references to remote directories and memories. Since the processor caches must be fast, they are generally not very large, so capacity misses vastly outnumber coherence misses. For each cache miss, the CC-NUMA must contact the line's home node, then retrieve a copy of that line, perhaps from a third node. Since this will typically involve crossing an interconnection network twice or more, the network needs to be very fast.

CC-NUMA machine performance will improve if directory entries and memory pages are local to the processor that needs them most frequently. When assignment of pages to nodes is sub-optimal the OS can try to move pages between nodes to minimize remote references. For a page with mostly read accesses, the OS might create replicas on different nodes, with remote procedure calls to other nodes to update or invalidate the multiple copies if any of the nodes write into the replicated page. These heuristics to improve performance involve significant software overhead, and may perform poorly. The grain size of data movement is a page, which will cause problems if multiple nodes modify different cache lines of the page. Statistics on cache line misses must be gathered in the interconnect hardware, increasing its complexity and cost.

Interconnect latency affects both coherence and capacity misses. As before, caching can be used to avoid trips over the interconnect and improve cache miss service times for capacity misses. Hence, some designs include an ***interconnect cache*** associated with the interface between a node and the interconnect network, so that more cache lines from remote nodes can be kept closer to the processor cache that might need them again. In effect, this is another level of caching in the memory hierarchy, and adds to the overall memory cost of the machine.

A CC-NUMA machine provides the programmer with the resources of many processors sharing a very large memory. It increases hardware cost with a fast interconnect network and hardware to maintain cache coherence in the interconnect cache. To assess the performance impact of the slower remote references, consider the time interval between remote accesses, measured in processor clock cycles. Call this the ***capacity miss interval***, M. During this time, the processor executes M/cpi instructions where cpi is the real processor clocks per instruction. Assume, as is typical, cpi is approximately 1.

If the remote access requires a time R to complete, the processor will require a time $M + R$ to complete M instructions. The effective processor speed will then be $M/(M+R)$, for a penalty of $1 - M/(M+R)$ or $R/(M+R)$. In the simple example where R is equal to M, the penalty is 50%. The CC-NUMA machine designer must minimize R, by increasing the interconnect speed, or raise M, by enlarging its interconnect or processor cache. Both these alternatives are very expensive.

Examples of CC-NUMA machines include the MIT Alewife[Ag91] and Stanford DASH[Le92] and FLASH[Ku94] machines. Machines built using the Scalable Coherent Interface (SCI) [Ie93] will also be CC-NUMA machines.

2.6 COMA

Kendall Square Research pioneered the ideas of a ***C***ache-***O***nly ***M***emory ***A***rchitecture where all memory in the machine is treated as cache memory. COMA is, in our opinion, the best way to build a cache-coherent, distributed-memory multiprocessor. In a COMA, virtual pages have no physical “homes”, and cache lines freely move about the machine as needed. The memory of each node, including the main memory, is allocated dynamically by cache line-replacement logic in hardware or software that controls the caches. Borrowing on the terminology of the Data Diffusion Machine[Wa88], we call this ***attraction memory***.

A COMA machine is technically a CC-NUMA machine as well, in that memory access times are non-uniform and caches are coherent. In this paper, we follow the common practice and use CC-NUMA to indicate the architecture of section 2.5, wherein each memory page is homed to one node of the machine.

The fundamental advantage of large attraction memories is that capacity misses in the attraction memory are almost eliminated. Once the system is operating at equilibrium, only coherence misses trigger memory requests among nodes. Capacity misses from processor caches always hit in the attraction memory. For a given job mix and node size, an attraction memory capacity miss is as unlikely as a main memory disk page miss in a UMA. The service time for the COMA miss will be a thousand times faster than a disk page fault if the necessary cache line can be found in the attraction memory of one of the other nodes in the machine.

A COMA node needing a cache line broadcasts its request through the interconnect to find a copy of the data. When updating shared data, a node must broadcast an invalidation to all nodes with read-only copies. Broadcast is necessary as the location of a needed cache line is not deterministic. The interconnect may store summary directory information for groups of nodes in the machine to help find needed cache lines and short-circuit invalidation messages.

COMA architectures use a protocol similar to that of UMA symmetric multiprocessors to maintain consistency among multiple caches. This allows COMA architectures to scale up from conventional (MESI) cache coherence architectures found in commodity PC servers today. The peers in this protocol are the nodes in the system, each of which may itself be a UMA machine. The nodes use directories to maintain state information about the cache lines in the shared virtual address space.

At any instant, the virtual address space of a system is partitioned among the nodes, with copies brought to the attraction memories where the running software needs them. The division depends on the processors' workload. Because the attraction memory functions as a very large cache, the COMA machine drives capacity miss rates, at the node level, very low. Once the flurry of initial cache misses that first brings in the program and data is complete, the system will see primarily coherence misses due to modifications of shared program data.

As a further benefit, for very large programs, the other nodes' memories are available to use for swap space, in effect, saving traffic to and from the system swap disk.

Because the traffic among COMA nodes is primarily for coherence, the interconnect can be optimized for coherence operations. The network should efficiently support delivery of invalidation requests to many nodes, and distribute copies of a popular cache line quickly following an invalidation.

For performance reasons, it is important that data movement and coherency management be done in hardware, at hardware speeds. Page management and replacement can reasonably be done in software, but software-based coherency management is slow. The granularity of data movement and coherency management should to be less than the page size, determined by sharing granularity and interconnect efficiency rather than disk paging considerations.

As with the CC-NUMA machine, we can compute the performance penalty of the COMA machine. The relevant measurements are the remote access time R, as before, and the interval between those accesses, which we call the ***coherence interval,*** C. This name reflects the observation that the COMA machine initiates remote accesses only upon coherence misses. The effective performance is, similarly, $C/(C+R)$. The COMA performs better, for a given R, because the coherence interval C is longer than the CC-NUMA miss interval, M. The penalty will be the same if the CC-NUMA machine has an interconnect that is C/M times as fast as the COMA, or if it adds sufficient cache memory to each node to make M equal to C. Section 4 shows that this is often a prohibitively large cache. Consequently, we expect COMAs to outperform comparable CC-NUMAs.

R is a function of the network and directory implementations, which are first order independent of the choice between COMA and CC-NUMA. For any R achieved by expending CC-NUMA engineering effort, the resulting network and directory implementation can be used by COMA, with the additional benefit of the large DRAM-based nodal memory available for caching shared data.

A CC-NUMA with software affinity algorithms that react perfectly to usage patterns in real time for all applications running can approach COMA, except that the granularity of the coherence unit would be a page instead of a cache line. Algorithms of this quality don't exist or are at least very costly to implement. Relative to proven hardware caching techniques, software algorithms are slow to infer affinity, require memory copying operations, and probably have to cope with maintaining coherence through software interrupts. COMA requires none of this to be done in software. Every needed page is brought to the local memory on first use and stays there unless coherence forces it invalid.

CC-NUMA could possibly approach COMA, if the affinity is determined by the application itself and if the application is amenable to affinity sets that are static long enough that the cost of memory copying is less than the cost of the capacity misses saved. But the copying is still slower than page replacement in COMA, since a CC-NUMA machine still has to bring over the remote page to make the copy, and has to cope with coherence events on the copy (by making it read-only and trapping writes to it). COMA does all this too, but at the hardware level without traps.

The KSR-1 and KSR-2 are both COMA machines [Ke92]. In 1987 Kendall Square Research patented these ideas and trademarked the term ALLCACHE® to describe them [Fr91]. COMAs also were investigated by research on the Data Diffusion Machine [Wa88], [Ha92]. Recently a group at the Stanford FLASH project has started an implementation of a COMA protocol in their interconnect processor[Ba95].

3 Cost of Multiprocessor Machines

The first large-scale multiprocessor machines were expensive to build and use. Their novel organizations necessitated special-purpose hardware plus modifications to the processor, memory and I/O systems. Software for these machines was typically non-standard and difficult to design and debug, both at the OS level and at the application level. The performance advantages of the early multiprocessor machines were accessible only to those who could afford an expensive computer and an expensive programming task. Many of the machines were super-computers, which could deliver high absolute computing power, but often for a small set of applications. In the end, the overall price per unit of computation was not competitive for many potential uses. Because the typical customers for these machines wanted the highest levels of performance, the machines were often built with exotic component technology. This drove their cost up and further limited their market.

At the same time, economies of scale and competitive pressures were making personal computers (PCs) cheaper. Their performance has risen to parity with other computers. PC servers have now reached performance levels available 10 years ago only in mainframe systems, while desktop systems are more powerful than the minicomputers of the 1980's. The price/performance of PCs is two to ten times better than that of minicomputers and mainframes.

The challenge regarding applicability to PCs is two-fold. The first is getting connectivity to the cache-coherent processor/memory bus. The second is delivering a software solution that is consistent with standard software - that is, a solution that can be layered on standard operating systems, without introducing scalablity or portability issues.

The important architectural features needed to build shared-memory multiprocessors are now appearing in all modern PC systems. Typical PC machines are being designed and configured with such enabling technologies as cache-coherent bus protocols and upgradable processor and memory subsystems. These UMA machines will represent the best price-performance point of all multiprocessors, and provide excellent performance up to their bus limits.

The PC cache coherence primitives and access to the processor-memory bus allow nodes to be interconnected as a multiple-node COMA machine. Cache misses can be routed to remote nodes and invalidates can be routed as necessary to invalidate cached copies. The interconnect coherence hardware can participate in the snooping protocol on the node's shared memory bus. The added hardware occupies upgrade slots already provided on the PC machine motherboards.

The requirement of COMA to attach to the processor/memory bus is no different from any other cache-coherent DSM approach - the best that can be done is to minimize the intrusion. COMA accomplishes this by introducing a snooping function and some state bits to the node, whereas CC-NUMA, to perform well on large problems, also requires another level of cache and the cache controller to drive it. The COMA approach need not interfere at all when the data is in the local memory. Memory transactions are snooped in parallel with the attached processors' caches.

This trend in the design and architecture of high-volume computer systems enables cost-effective COMA computer systems. Soon, the cheapest multiprocessor designs will be built using hardware added to commodity PC machines for a small incremental cost. Most of the hardware in these COMA machines will stay on the industry's learning curve and benefit from high-volume production economics.

Concurrent with these advances in commodity hardware, complementary enabling technologies have arisen in software architecture. Modern operating systems provide modularity and dynamic configurability, including the ability to plug in third-party modules for such important components as device drivers, and run-time libraries that provide standard program services for formatted I/O and interprocess communication among user applications. Any user application capable of multiple-process or multiple-threaded operation already has the necessary organization to take advantage of the aggregated resources of the combined COMA machine.

These advances are available to all multiprocessor system designers. In particular, both of the preferred architectures, CC-NUMA and COMA, can today be built from components that add a small fraction, on the order of 10%, to the cost of the processing nodes and their software to combine them into a larger cache-coherent multiprocessor. Moreover, the nodes themselves can be bought from the same high-volume systems manufacturers that today provide the best price-performance figures.

For a given investment in interconnect hardware, the COMA architecture will always provide superior performance. The advantage flows from its ability to use all or part of a node's main memory as an attraction memory to hold shared data needed by the processes running on the node. If the node's memory is insufficient for those processes, one can easily add more commodity DRAM memory up to the limits of its design.

Once configured properly for its operating point, a COMA machine ought to observe inter-node memory traffic due only to coherence misses. The large attraction memories minimize capacity misses. By contrast, eliminating CC-NUMA capacity misses would require huge interconnect caches, which would be expensive. Alternatively, to decrease the interconnect latency and minimize the effect of capacity misses, the CC-NUMA machine would require a faster and costlier interconnect.

One could view a CC-NUMA machine as a COMA machine with a fixed division between private memory and cached, shared memory. What should the relative memory sizes be? Since the answer depends on the workload, there is no single answer. Therefore, COMA's ability to trade private memory for shared memory dynamically should yield better performance over a wide range of applications. In addition, it can match the performance at the operating point where the CC-NUMA interconnect cache is sized properly.

Thus COMA is a functional superset of CC-NUMA: COMA makes all of a node's DRAM available as an interconnect cache dynamically, as needed, if needed. Relative to software approaches, it accomplishes data affinity and coherency management at hardware speeds, a useful feature whether or not one has additional data partitioning, affinity or replication approaches implemented in software, and whether or not one chooses to use the full power of shared memory (because of availability or scalability concerns). COMA is compatible with various node interconnect strategies and software approaches. It can be implemented for less cost than CC-NUMA and requires less logic proximate to the processor/memory bus.

The best price-performance available for the coming generations of processor chips will be UMA machines up to the capacity of a single bus. Beyond that, for a small penalty in price-performance, on the order of 10% or less, COMA machines will allow growth beyond the single-bus limit to total system performance an order of magnitude higher and more. Both CC-NUMA and COMA machines will continue to scale well with new processor generations and chip technologies.

4 Three COMA Case Studies

To characterize the behavior of COMA for typical commercial database applications, we examined three fundamental operations: OLTP, sorting, and database join operations. The latter two workloads are typical of commercial batch and decision support (data mining) applications. These workloads provide quantitative estimates of remote memory access frequency (coherence miss interval *C*) in a variety of database workloads using Oracle7 Parallel Query Option. Together with latency and bandwidth predictions from hardware modeling, they can help predict COMA performance and scalability.

This section shows that COMA exhibits two or three times fewer cache misses than CC-NUMA on three sample problems. This is a HUGE performance advantage. The advantage will be even greater as systems scale to more nodes or faster processors.

4.1 Hardware and Software Configuration

We conducted the experimental work on a KSR-1 [Ke92]. The KSR-1 and KSR-2 are the only current commercial implementations of COMA. The application DBMS was Oracle7. Each node of a KSR-1 has a 20 MHz RISC processor and a 32 MB attraction memory. The nodes of the KSR-1 are arranged in a hierarchy of rings or AllCache Groups, with 32 nodes on each of the lowest AllCache Groups (AG:0). Up to 34 AG:0 groups can be connected on a second ring or AG:1, for a maximum of 1088 nodes.

The remote access latency of an attraction memory miss in the KSR-1 averages about 150 cycles between two nodes on the same AG:0. The latency is about 570 cycles between nodes on separate AG:0 rings, because of the additional time needed for packets to travel over the AG:1 ring and the remote AG:0 ring.

Since the development of the KSR-1, processor speeds have increased more rapidly than interconnect latencies have decreased, so we expect that the time for remote accesses, as measured in clock cycles has increased. Accordingly, the performance loss caused by short average intervals between misses will be relatively greater and the need to minimize the ratio of remote access time *R* to average miss interval *C* will be even greater.

Compared to the KSR-1, nodes of future COMA machines will have several much-faster processors and much more attraction memory, up to 1 GB or more. They will also have faster interconnect capability.

We focused on measurement of the coherence interval, because most remote accesses in a COMA machine are coherence misses. Because a single remote access takes a long time (200 or more processor cycles), the coherence interval *C* should be long compared to the remote access latency *R* in order to minimize the impact on performance, $R/(R+C)$. We generally observed a coherence interval of 500 to 2000 processor cycles. This is very promising. It suggests that a COMA will have a small (10% to 25%) performance penalty relative to ideal linear speedup.

Because a KSR-1 node with a 32 MB attraction memory is not a close model of a COMA machine node with several modern microprocessors and lots of memory, and because the KSR-1 interconnect was slower than modern technology allows, we must take care to interpret the results for future COMA machines.

4.2 On-Line Transaction Processing (OLTP)

Transaction processing performance (particularly the TPC-B benchmark) is one of the metrics commonly used for comparing computer platforms [Gr93]. The workload presents a large number of concurrent threads, each executing a relatively simple transaction as fast as possible. We used a TPC-B-like application executing on a KSR-1 to measure the frequency of remote accesses. We also enumerated system calls in each transaction that would generate internode traffic in a COMA machine and estimated their contribution to remote accesses.

A 32-node KSR-1 achieved approximately 300 TPS, about 25 times the speed of a single node. More importantly, we measured the remote access rate. Most of the remote accesses are generated by Oracle's concurrency control "latches" and the native synchronization primitives of the hardware. Oracle7 does not use OS system calls for this.

The Oracle latch implementation is architecture dependent. The typical port of Oracle on a modern processor would use some variant of an atomic "fetch and operate" instruction such as "test-and-set". We experimented with several latch manipulation implementations and found that the fastest implementation on a KSR-1 did not necessarily yield the fewest attraction memory misses because TPC-B performance on a KSR-1 is limited by processor speed more than by remote memory latency.

Choosing the latch manipulation algorithm that minimized attraction memory misses resulted in about 350 remote accesses per transaction over a wide range of concurrency and aggregate speed. We think that this number is a reasonable estimate of the coherence misses per transaction that a modern COMA or CC-NUMA machine would experience, including the remote accesses related to system calls that request resources on remote nodes (for example, reads, writes, semaphores). If we assume a 100 MIPS processor can perform 150 transactions per second, then the coherence interval will be about 2000 instructions per remote access [(100 x 10^6 instructions/sec) / (150 transactions/sec) / (350 remote accesses/transaction) = 1900 instructions/remote access], which is much larger than the latency of a single remote access.

In a separate detailed report [Hi95], Hill analyzes the impact of those coherence misses on TPC-B performance for a COMA machine with several fast microprocessors per node (see Figure 1). The key conclusions of that report are:

- Relative to ideal linear speedup, hardware scalability would be about 85% for two nodes and 75% for 8 nodes.
- Bandwidth and latency requirements can be met economically with current technology.
- Expensive efforts to lower interconnect latency dramatically would not be cost effective.
- System performance is sensitive to the frequency of remote accesses.

The last point emphasizes the importance of limiting remote accesses to coherence misses. If the interconnect cache of a CC-NUMA machine is not large enough to effectively eliminate capacity misses, then the extra expense of a very fast interconnect is required to neutralize the performance impact of the extra remote accesses. The attraction memory of a COMA machine node will be large enough that it will suffer no capacity misses.

4.3 Sorting

Sorting is a typical database operation in a Decision Support System (DSS). For example, it is used both for creating indices and in the sort/merge form of the database join operation. Sorting is "hard" for distributed systems, including COMA, because nearly all of the data must move from one node to another during the sort, thus it is not an example of an intrinsically parallel operation.

In this case study, we used Oracle Parallel Query Option (PQO)) to sort a moderately large table (up to 1,000,000 rows of 100 bytes each, 100MB) as part of a sort/merge join query. PQO uses separate processes to partition and sort a large table.

For P-way parallelism, P processes scan the data in parallel and divide them into P partitions, based on the values of the sort key. P separate parallel processes sort the P partitions. Most of the computation takes place in the sort processes.

4.3.1 Inter-node traffic

Ideally, most of the inter-node traffic will be in the transport of data from the scan processes to the sort processes. In general, those processes will be on separate nodes. Unless the scan processes can be forced onto nodes that are locally connected to the disks that contain the data they scan (I/O affinity), there will also be internode traffic from the I/O nodes to the scan nodes. In our study, we did not configure the system with I/O affinity, so our analysis includes the I/O related traffic.

4.3.2 Inter-node latency

We found that, given quantitative assumptions about the performance of a typical modern COMA machine, the remote access latency due to coherence misses adds about 10% to the time required for the sort processes to sort their partitions and that bandwidth requirements can be met economically.

Based on measurements of PQO on a KSR-1, Oracle7 uses approximately 530 cycles * N * $\log_2(N)$ to sort N rows and causes 3 * N remote accesses. The cycles per attraction memory miss are about 530/3 * $\log_2(N)$, for a miss interval of about 3,500 cycles, if N = 1,000,000 = 2^{20}. Assuming a typical miss latency and CPU speed of a modern COMA machine, each miss costs 400 cycles, degrading the CPU performance by ~10%. The cycles for one process to sort 1M rows (100B/row) would be

$$T = 530 \text{ cycles} * 1M * 20 + 400 \text{ cycles} * 3 * 1M = 10.6\text{x}10^9 \text{ cycles} + 1.2\text{x}10^9 \text{ cycles} \sim= 12\text{x}10^9 \text{ cycles.}$$

The coherence interval, access time, and efficiency, as defined in section 2.6, are

$$C = 3{,}500,$$

$$R = 400,$$

and

$$C/(C+R) = 0.90.$$

4.3.3 Bandwidth requirements

Suppose each node has four processors executing 3 sort processes and 3 scan processes. The sort processes are CPU intensive and would use one processor each. The scan processes are not CPU intensive and could share a processor. Assume each sort process has 1M rows to sort and takes $12\text{x}10^9$ cycles to do it. From the scan processes, the three sort processes pull 3 misses/row * 128B/miss * 3M rows = 1.2GB to the node in $12\text{x}10^9$ cycles, i.e., $100\text{MB}/10^9$ cycles.

When the scan processes write into the buffers they use to communicate with the sort processes, they pull the same cache lines back, contributing another $100\text{MB/x}10^9$ cycles. Assuming no affinity between the scan processes and the I/O nodes, the three scan processes must also pull the original data from the I/O nodes, or 300MB in $12\text{x}10^9$ cycles or $25\text{MB/x}10^9$ cycles.

The total in-bound traffic to the node is 2 * $100\text{MB/x}10^9$ cycles + $25\text{MB/x}10^9$ cycles ~= $225\text{MB/x}10^9$ cycles. Assuming a cycle speed of 100MHz yields 23MB/second coming into each node, well within the bandwidth capability of modern economical hardware. Conservation of remote accesses tells us that the out-bound traffic at each node will be equal to the in-bound traffic.

4.3.4 Sensitivity

Suppose that a next-generation processor had a cpi half as large and the clock rate were twice as fast as assumed above. Then the cycles for one process to sort 1M rows (100B/row) would be

$$T = (265 \text{ cycles}) * 1M * 20 + 800 \text{ cycles} * 3 * 1M = 5.3x10^9 \text{ cycles} + 2.4x10^9 \text{ cycles} = 7.7x10^9 \text{ cycles}.$$

The coherence interval, access time, and efficiency are

$$C = 1750,$$

$$R = 800,$$

and

$$C/(C+R) = 0.69.$$

This is about 2/3 as many cycles as the base case in section 4.3.2 and 3 times as fast (because the cycle speed is assumed to be a factor of 2 faster). The network traffic would be about 70MB/second, which is much closer to the limit of modern economical hardware. This introduces queuing delays and requires a more sophisticated model to get a quantitatively correct result, so the above estimate is optimistic.

4.3.5 Capacity misses of COMA vs. CC-NUMA

Each sort process uses a large, unshared sort area to sort its partition in memory. That sort area is likely to be large compared to any interconnect cache on the node of a CC-NUMA machine. If a CC-NUMA machine were not carefully coded to use local memory for each sort process, then the sort processes would suffer significant capacity misses in the small interconnect cache.

As an example, we estimate the cache capacity misses in a 1,000,000 row sort, assuming that the sort area used by the sort process is allocated globally and not specifically on the local node of the sort process.

The observed user code attraction memory misses of PQO sort processes on a KSR-1 are approximately 3 misses per 100B row for 10,000 <= N <= 50,000 and significantly greater than that for N >= 50,000, where N is the number of rows to be sorted by each process. We interpret the attraction memory misses below 50,000 rows per sort process to be initialization misses retrieving data from the scan processes and we interpret the excess misses for N > 50,000 to be capacity misses when the data are too large to be sorted in local attraction memory of the KSR-1.

Assume that each sort process builds a binary tournament tree of (key, pointer) pairs with (key, row) pairs at the leaves of the tree [Kn73]. Elements are added to the tree as new rows are received from the sort processes. Here is a rough, simple, analytical model for predicting cache misses as new elements are added.

Let

n = the number of elements in the tree,

c = the number of elements in the tree when the cache is effectively full,

a(n) = the number of tree element visits after inserting n elements,

m(n; c) = the number of tree element visits that generate capacity misses inserting n elements.

On each insertion, a path through the tree of length $\log_2(n)$ has to be traversed, so:

$da/dn = \log_2(n)$

and

$a(n) = n * (\log_2(n) - \log_2(e))$

where ‘e’ is the base of the natural logarithms.

Because each tree node visit accesses more memory (e.g., code and data) than just the memory representing the tree node, we expect the number of memory accesses to be larger than the number of tree nodes visited. On the KSR-1, we have measured the attraction memory accesses and the CPU busy cycles and both are very nearly proportional to $n * \log_2(n)$ as expected. The ratios are about 10.5 memory accesses and 530 CPU cycles per tree node visited.

To model the attraction memory capacity misses, assume $m(n; c) = 0$ for $n < c$, meaning that there are no misses until the effective cache is full and that

$dm/dn = \log_2(n) - \log_2(c)$ for $n >= c$

meaning that the first $\log_2(c)$ elements in the path are cached and none of the rest of the elements in the path are cached.

Then

$m(n;c) = 0 \quad ; n <= c$

$m(n;c)= (n * \log_2(n/c)) - ((n - c) * \log_2(e)) \quad ; n >= c$

In the assumptions, we have ignored the size of the cache lines and the element alignment, which would sometimes cause more than one miss on a single retrieval and sometimes retrieve more than one element in a single retrieval. We have also ignored the effect of the different sizes of internal and leaf nodes in the tree, and the effect of other data and code using the cache.

For example, let c = 100,000 and n = 100,000, 500,000 and 1,000,000. For 100B rows, c = 100,000 implies that the effective cache size for a single sort process is about 10MB. Assuming 3 or 4 sort processes are sharing a single node, the attraction memory would have to be at least 30MB - 40MB and probably more to accommodate caching other data too. These results are summarized in Table 1.

rows inserted	**capacity misses**
100,000	0
500,000	580,000
1,000,000	2,020,000

Table 1. Capacity misses suffered inserting rows into a tournament tree, assuming an interconnect cache large enough to hold 100,000 rows (approximately 10MB - 20MB) before incurring any capacity misses.

The measurements on a KSR-1 indicate that there are about 3 initialization misses per row. If each of the node misses leads to three capacity misses in practice when each sort process has to sort one million rows (n = 1,000,000), then the capacity misses for this example would be about twice the initialization misses (i.e., 3,000,000 initialization misses plus 6,000,000 capacity misses). The small (e.g., 10MB - 20MB) interconnect cache CC-NUMA machine would have about three times the total misses relative to a COMA machine wherein all attraction memory is available for effective caching.

4.4 Nested Loop Join

In a nested loop join, the DBMS uses two tables, an “outer” table and an “inner” table. It scans each row in the outer table, and, for each row in the outer table, looks for a corresponding row in the inner table, based on the value of a common key, using an inner table index based on the key.

At each node, COMA would automatically cache and replicate as many inner table index and data blocks as would be useful. CC-NUMA could only cache and replicate as many blocks as fit in the interconnect cache. A fixed size interconnect cache would not adapt to the needs of the application and would usually be too small to avoid capacity misses.

As before, the key metric is the miss interval. In this case, we are interested in the number of additional remote accesses that a CC-NUMA machine would incur because its interconnect cache is smaller than the attraction memory of a COMA machine.

Of the three case studies, only this one is not based on measurements of a KSR-1, which was not available at the time of the study. Furthermore, the 32 MB attraction memory of the KSR-1 would not be large enough to hold a copy of a large table index at each node, as discussed below. Modern COMA machines could have such large attraction memories, and thus would have a significant advantage over the KSR-1.

4.4.1 Analysis

Consider a concrete example, joining two tables on some key that is common to both tables. The outer table is scanned sequentially. At each row of the outer table, the inner table is probed via a B-tree index using the key value from the outer table. On each probe of the inner table, how many CPU cycles are executed and how many memory accesses are remote?

Oracle uses a large shared memory area called the ***S***ystem ***G***lobal ***A***rea (SGA). Aside from shared variables, the primary use of the SGA is for buffers for disk blocks from the database while they are needed by database operations. Oracle manages the buffers as a data block cache with ***L***east ***R***ecently ***U***sed (LRU) semantics.

As an illustrative example, consider a nested loop join with the following inner table parameters, for comparison between COMA and CC-NUMA:

Oracle SGA = 64K blocks * 16KB/block = 1GB
Physical memory >= 1 GB per node.

Index:	root level:	256KB	= 16 blocks
	second level:	16MB	= 1K blocks
	leaf level:	1GB	= 64K blocks
inner table:		64GB	= 4M blocks

The SGA is large enough to keep all the root and second level index blocks in the SGA and nearly all the leaf-level blocks. To take advantage of the large SGA, the cached leaf level blocks must be used more than once. Since each row of the outer table generates a single probe through the index, the leaf blocks won’t be used a second time on average unless there are >64K rows in the outer table, or 8MB = 512 blocks at 128 B/row.

A moderate size interconnect cache (~16MB per node) is large enough for a CC-NUMA machine to keep copies of the first two index-level blocks at each node. (Actually, the interconnect cache will have to be substantially larger than that in order to hold the working sets of all the processes executing on the node plus the data working set of the index.) Each probe will require reading leaf-level index data from a remote node. But a COMA machine can keep a copy of the entire index at each node.

Both machines have to read an inner table data block from disk on each probe. This will cause evictions from the Oracle buffer cache. In the COMA machine, the resulting attraction memory activity will be limited to the coherence misses implied by reuse of the SGA buffer. In a CC-NUMA machine, the data buffer reference will have the further effect of evicting some other cache line, typically a second index block which will be needed again, for a capacity miss. For simplicity, we ignore this effect in the following.

How many extra remote accesses does a CC-NUMA machine use to read the leaf-level index data and how does that compare to the cycles required for a COMA machine to do the probe?

Each 16KB block of the index would contain a hundred or more shared memory cache lines (for example in the KSR-1, attraction memory cache lines are 128 bytes each). To get the address of the inner table data block, the probe will require a binary search of the index block, i.e., ~7 cache lines. Because index records won't generally be aligned to the cache lines, we estimate that 10-15 cache lines from the leaf index block will be read remotely on CC-NUMA, but theses accesses will be local with COMA.

Both architectures need to read an inner table data block from disk into the SGA. That will require 5-10 cache lines for coherence misses to allocate an SGA buffer block. Thus the CC-NUMA machine will have 15-25 total remote accesses while the COMA machine will have 5-10 total remote accesses, giving COMA a 2.5:1 or greater advantage over CC-NUMA in terms of remote memory accesses. Because the CC-NUMA has 2.5 times as many remote accesses, the COMA miss interval is 2.5 times longer than the CC-NUMA miss interval.

We expect that the length of the code path in the probe will be dominated by the operating system code to read the data block from disk. Based on measurements of the KSR-1, we assume that this code path length is approximately 20K cycles. Thus 20K cycles per probe divided by 15 - 25 remote accesses per probe yields a miss interval of 800 - 1300 cycles for CC-NUMA and the same path length divided by 5 - 10 remote accesses per probe yields a miss interval of 2000 - 4000 cycles for COMA. If we assume a remote access latency 400 cycles, the cost of the remote accesses for CC-NUMA is 25% to 35% (1300/(1300 + 400) = 0.77 and 400/(800 + 400) = 0.67). For COMA, the cost is only 10% to 20% (4000/(4000 + 400) = 0.91 and 1000/(2000 + 400) = 0.83).

If the inner table is small enough to be buffered completely in the Oracle SGA along with all its index blocks, then the comparative advantage of COMA is even greater. Buffering most blocks reduces I/O, decreasing the average number of code execution cycles per probe. But the number of remote accesses stays the same, decreasing the miss interval. If the miss interval decreases to M = 400 cycles for CC-NUMA and C = 800 cycles for COMA, then COMA will be 33% faster than CC-NUMA (1200 clock cycles vs. 1600 clock cycles to complete 800 instruction cycles.)

4.4.2 Sensitivity

The analysis is not particularly sensitive to the fanout ratio of the index. The above discussion assumed each index block contained addresses for 64 blocks in the next level down. If we assume a ratio of 128 instead, then we have:

Index:	root level:	32KB	= 2 blocks
	second level:	4MB	= 256 blocks
	leaf level:	512MB	= 32K block
inner table:		64GB	= 4M block

Or we can assume a 4x larger database:

Index:	root level:	16KB	= 1 block
	second level:	1MB	= 64 blocks
	third level:	64MB	= 4K blocks
	leaf level:	4GB	= 256K block
inner table:		256GB	= 16M block

In the larger database, both machines would have to go to disk for some of the leaf index blocks in addition to the table data blocks, but the CC-NUMA would have to make remote references for some of the third level index blocks in addition to the cached leaf level blocks.

4.4.3 Large Oracle SGA

If the Oracle SGA could be specified to be larger than the physical memory of a single node, some of the COMA advantage would decrease because the COMA machine would suffer some capacity misses on references to SGA buffers that are not cached locally. In general, a large SGA is best because it minimizes disk I/O when data blocks are not found cached in the SGA buffer pool. The COMA address space would be better used by caching more of the index and table in shared memory instead of replicating all buffer blocks of smaller SGA on every node.

Currently, however, the SGA can have a maximum of 64K blocks, effectively limiting it to about 1GB of 16KB blocks. In principle, SGA could be made larger. In December, Digital demonstrated an Alpha/OSF Oracle with an 8GB SGA using 32KB buffers and 256K buffer blocks. That requires a 64 bit addressing architecture and a large virtual memory allocation to a single process. Many current architectures and operating systems are limited to 32 bit addresses and 1-2GB of virtual memory per process. That imposes an effective upper bound on the Oracle SGA for the near future, allowing each node in a COMA machine to map the entire SGA into its local physical memory.

5 Summary and Conclusions

5.1 Multiprocessor architectures.

We summarized the development of multiprocessor computers with distributed memory and presented three case studies that illustrate the strength of Cache-Only Memory Architecture or COMA.

We reviewed the history of computer architectures from the viewpoint of memory hierarchies and shared memories. Virtual memory allows programs to be written as if the available physical memory is much larger than the actual physical memory. A relatively small but fast memory cache allows programs to execute as if the physical memory were much faster than it really is. The cache, physical memory, and backing store of the virtual memory form a memory hierarchy.

Cache misses are an important determinant of system performance. We categorized misses into initial, capacity, and coherence misses.

We described several multiprocessor memory architectures. No Remote Memory Access (NORMA) multiprocessors do not directly share memory among processors, but exchange data by passing messages. They are the least expensive, but are difficult to program. Uniform Memory Access (UMA) multiprocessors use a single bus to connect the processors and memory. As processors become faster, the finite speed of the shared bus limits the number of processors that can efficiently share memory. Non-Uniform Memory Access (NUMA) multiprocessors consist of multiple nodes, with each node containing one or more processors and some local memory. An interconnect mechanism allows the nodes to refer to the memory of other nodes, but a reference to a remote node takes longer than a local reference. Caches may be used to speed up memory references, but software protocols are required to maintain cache consistency.

Cache Consistent NUMA (CC-NUMA) multiprocessors use hardware to maintain cache consistency. The performance cost is $M/(M+R)$ where M is the interval between misses and R is the cost of a miss. An additional cache level or interconnect cache may lower the rate of remote memory references.

A Cache-Only Memory Architecture (COMA) multiprocessor also consists of multiple nodes, each with one or more processors and some local memory. But in the COMA machine, data addresses are not assigned to fixed physical nodes. All the physical memory in each node is treated as cache, and is referred to as attraction memory. Copies of data move freely among the nodes as needed. The effect is similar to that of a CC-NUMA with a large enough interconnect cache to eliminate capacity misses.

As with a CC-NUMA multiprocessor, the performance cost of a remote references is given by $C/(C+R)$, where R is the remote access time and C is the average interval between remote accesses, or coherence interval. The COMA coherence miss interval is much larger than the miss interval in a CC-NUMA, which yields a significant performance advantage.

5.2 Costs

Commodity personal computers have better price/performance than other computers, and are now appearing in shared memory multiprocessor UMA configurations. These commodity components can be connected with inexpensive "add-in" hardware to convert them to COMA (or CC-NUMA) multiprocessors, resulting in processor counts and performance levels that are unavailable to UMA multiprocessors, with nearly the price/performance of the commodity products. Modern system software allows "add-in" software to take advantage of the high-performance, low-cost hardware.

COMA eliminates the expense and complication of the additional interconnect cache required by CC-NUMA and allows the physical memory to be dynamically divided between shared and private memory, adapting to the needs of the application load. For a given interconnect latency and cache directory size, COMA and CC-NUMA hardware introduce similar costs to build a larger, cache-coherent multiprocessor. The COMA machine's advantage in lower remote memory access rates gives it a significant performance advantage.

5.3 Case studies

We reported the results of three case studies based on measurements taken from Oracle7 and the KSR-1, the only current commercial implementation of COMA. The analysis of these studies applies to COMA designs with multiple processors at each node, and to faster processors and more memory per node than in the KSR-1.

5.3.1 OLTP

Transaction processing is implemented with a large number of concurrent threads, each executing a relatively simple transaction. We measured the coherence interval to be over 3000, leading to a prediction of hardware scalability of 75% for up to 32 modern microprocessors. A CC-NUMA multiprocessor without a large enough interconnect cache would have a shorter miss interval, reducing the scalability significantly.

5.3.2 Sorting

The Oracle Parallel Query Option (PQO) uses separate processes to partition and sort data in parallel, requiring almost all data to be moved at least once from a source node to destination node, with a coherence interval of about 3,500. The latency and bandwidth requirements for this inter-node traffic are well within the capability of economical modern hardware. Processes on the destination nodes that do the final sort of each partition need to allocate a large sort area in process private memory. If that sort area is not local to the node of the sorting process, many needless remote memory references will be generated in CC-NUMA. The COMA attraction memory automatically avoids those unnecessary remote references without any special programming effort. COMA had three times fewer cache misses than CC-NUMA.

5.3.3 Nested Loop Join

In a nested loop join, the DBMS scans each row in an outer table, and looks for a row in an inner table with a matching key value, using an inner table index. The typical coherence interval is 1000-4000. At each node, COMA would automatically cache and replicate inner index and data blocks. CC-NUMA can only cache and replicate as many blocks as fit in the interconnect cache. Typically each probe in a CC-NUMA multiprocessor would generate twice as many remote accesses as a COMA multiprocessor. COMA had two times fewer cache misses than CC-NUMA.

5.4 Conclusions

We conclude that COMA is the best way to build a cache-coherent, distributed-memory multiprocessor for commercial database processing. The COMA nodes can be implemented from high-volume systems that provide the best price-performance of any computers available, and COMA allows scalable distributed memory multiprocessors with higher processor counts than other designs. For a given investment in interconnect hardware, the COMA architecture will always provide superior performance. In the cases we examined it had two or three times fewer remote memory accesses -- a huge performance advantage.

6 References

[Ag91] Agarwal, Anant, et al. The MIT Alewife Machine: A Large-Scale Distributed-memory Multiprocessor. MIT/LCS Memo TM-454, Massachusetts Institute of Technology, Cambridge, MA, 1991.

[Ba95] Baxter, Joel. FLASH Architecture Status. Stanford, CA, January, 1995

[Cr93] Cray Research, Inc. Cray T3D System Architecture. 1993.

[Ha92] Hagersten, Erik. *Toward Scalable Cache Only Memory Architectures*. Ph.D. thesis, Swedish Institute of Computer Science. Kista Sweden, October 1992.

[Fr91] Frank, Steven J., et al. Multiprocessor Digital Data Processing System. US Patent 5,055,999. Filed December, 1987; granted October, 1991.

[Gr93] *The Benchmark Handbook for Database and Transaction Processing Systems*, Gray, J. (ed.), *Chapter 3: TPC-B Specification*, Morgan Kaufmann, San Francisco, 1993.

[Hi95] Hill, Tim, AllCache SuperServer Performance Model, Kendall Square Research, 26 January 95

[Ie93] IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface. IEEE Computer Society, New York, NY, August 1993.

[In91] Intel Corporation. Paragon XP/S Product Overview. 1991.

[Ke92] Kendall Square Research Corporation. KSR1 Technical Summary. Waltham, MA, 1992.

[Ki62] Kilburn, T., et al. One-level storage system. IRE Transactions on Electronic Computers EC-11, pp 223-35. April, 1962.

[Kn73] Knuth, Donald E. *The Art of Computer Programming, Volume. 3*, Addison-Wesley, 1993.

[Ku94] Kuskin, Jeffrey, et el. The Stanford FLASH Multiprocessor. In Proceedings of the 21st International Symposium on Computer Architecture, pp 302-313. Chicago, IL, April 1994.

[Le92] Lenoski, Dan, et al. The Stanford DASH Multiprocessor. IEEE Computer, 25(3), pp 63-79, March 1992.

[Ma94] Maynard, Ann Marie Grizziffi, et al. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), pp 145-155. 1994.

[St86] Stonebraker, M., The case for shared-nothing. IEEE Database Eng., vol. 9, no. 1. March, 1986.

[St92] Stenstrom, Per, et al. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In Proceedings of the 19th Annual International Symposium on Computer Architecture, pp 80-91. Gold Coast, Australia, May, 1992.

[Th92] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary. Cambridge, MA, January, 1992.

[Wa88] Warren, David H. D. and Seif Haridi. The Data Diffusion Machine - A Scalable Shared Virtual Memory Multiprocessor. In Proceedings of the 1988 International Conference on Fifth Generation Computer Systems, pp 943-952. Tokyo, Japan, December 1988.