Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures

Per Stenström†, Truman Joe, and Anoop Gupta

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Two interesting variations of large-scale shared-memory machines that have recently emerged are *cache-coherent non-uniform-memory-access* machines (CC-NUMA) and *cache-only memory architectures* (COMA). They both have distributed main memory and use directory-based cache coherence. Unlike CC-NUMA, however, COMA machines automatically migrate and replicate data at the main-memory level in cache-line sized chunks. This paper compares the performance of these two classes of machines. We first present a qualitative model that shows that the relative performance is primarily determined by two factors: the relative magnitude of capacity misses versus coherence misses, and the granularity of data partitions in the application. We then present quantitative results using simulation studies for eight parallel applications (including all six applications from the SPLASH benchmark suite). We show that COMA's potential for performance improvement is limited to applications where data accesses by different processors are finely interleaved in memory space and, in addition, where capacity misses dominate over coherence misses. In other situations, for example where coherence misses dominate, COMA can actually perform worse than CC-NUMA due to increased miss latencies caused by its hierarchical directories. Finally, we propose a new architectural alternative, called COMA-F, that combines the advantages of both CC-NUMA and COMA.

1 Introduction

Large-scale multiprocessors with a single address-space and coherent caches offer a flexible and powerful computing environment. The single address space and coherent caches together ease the problem of data partitioning and dynamic load balancing. They also provide better support for parallelizing compilers, standard operating systems, and multiprogramming, thus enabling more flexible and effective use of the machine. Currently, many research groups are pursuing the design and construction of such multiprocessors [12, 1, 10]. As research has progressed in this area, two interesting variants have emerged, namely CC-NUMA (cache-coherent non-uniform memory access machines) and COMA (cache-only memory architectures). Examples of the CC-NUMA machines are the Stanford DASH multiprocessor [12] and the MIT Alewife machine [1], while examples of COMA machines are the Swedish Institute of Computer Science's Data Diffusion Machine (DDM) [10] and Kendall Square Research's KSR1 machine [4].

Common to both CC-NUMA and COMA machines are the features of distributed main memory, scalable interconnection network, and directory-based cache coherence. Distributed main memory and scalable interconnection networks are essential in providing the required scalable memory bandwidth, while directory-based schemes provide cache coherence without requiring broadcast and consuming only a small fraction of the system bandwidth. In contrast to CC-NUMA machines, however, in COMA the per-node main memory is converted into an enormous secondary/tertiary cache (called *attraction memory* (AM) by the DDM group) by adding tags to cache-line sized chunks in main memory. A consequence is that the location of a data item in the machine is totally decoupled from its physical address, and the data item is automatically migrated or replicated in main memory depending on the memory reference pattern.

The main advantage of the COMA machines is that they can reduce the average cache miss latency, since data are dynamically migrated and replicated at the main-memory level. However, there are also several disadvantages. First, allowing migration of data at the memory level requires a mechanism to locate the data on a miss. To avoid broadcasting such requests, current machines use a hierarchical directory structure, which increases the miss latency for global requests. Second, the coherence protocol is more complex because it needs to ensure that the last copy of a data item is not replaced in the attraction memory (main memory). Also, as compared to CC-NUMA, there is additional complexity in the design of the main-memory subsystem and in the interface to the disk subsystem.

Even though CC-NUMA and COMA machines are being built, so far no studies have been published that evaluate the performance benefits of one machine model over the other. Such a study is the focus of this paper. We note that the paper focuses on the relative performance of the two machines, and

† Per Stenström's address is Department of Computer Engineering, Lund University, P.O. Box 118, S-221 00 LUND, Sweden.

not on the hardware complexity. We do so because without a good understanding of the performance benefits, it is difficult to argue about what hardware complexity is justified.

The organization of the rest of the paper is as follows. In the next section, we begin with detailed descriptions of CC-NUMA and COMA machines. Then in Section 3, we present a qualitative model that helps predict the relative performance of applications on CC-NUMA and COMA machines. Section 4 presents the architectural assumptions and our simulation environment. It also presents the eight benchmark applications used in our study, which include all six applications from the SPLASH benchmark suite [14]. The performance results are presented in Section 5. We show that COMA's potential for performance improvement is limited to applications where data accesses by different processors are interleaved at a fine spatial granularity and, in addition, where capacity misses dominate over coherence misses. We also show that for applications which access data at a coarse granularity, CC-NUMA can perform nearly as well as a COMA by exploiting page-level placement or migration. Furthermore, when coherence misses dominate, CC-NUMA often performs better than COMA. This is due to the extra latency introduced by the hierarchical directory structure in COMA. In Section 6, we present a new architectural alternative, called COMA-F (for COMA-FLAT), that is shown to perform better than both regular CC-NUMA and COMA. We finally conclude in Section 7.

2 CC-NUMA and COMA Machines

In this section we briefly present the organization of CC-NUMA and COMA machines based on the Stanford DASH multiprocessor [12] and the Swedish Institute of Computer Science's Data Diffusion Machine (DDM) [10]. We discuss the basic architecture and the coherence protocols, directory structure and interconnection network requirements, and finally the software model presented by the architectures.

2.1 CC-NUMA Architecture

A CC-NUMA machine consists of a number of processing nodes connected through a high-bandwidth low-latency interconnection network. Each processing node consists of a high-performance processor, the associated cache, and a portion of the global shared memory. Cache coherence is maintained by a directory-based, write-invalidate cache coherence protocol. To keep all caches consistent, each processing node has a directory memory corresponding to its portion of the shared physical memory. For each memory line (aligned memory block that has the same size as a cache line), the directory memory stores identities of remote nodes caching that line. Thus, using the directory, it is possible for a node writing a location to send point-to-point messages to invalidate remote copies of the corresponding cache line. Another important attribute of the directory-based protocol is that it does not depend on any specific interconnection network topology. Therefore, any scalable network, such as a mesh, a hypercube, or a multi-stage network, can be used to connect the processing nodes.

Handling a cache miss in a CC-NUMA machine requires knowledge about the *home node* for the corresponding physical address. The home node is the processing node from whose main memory the data is allocated. (It is usually determined by the high-order bits of the physical address.) If the local node and the home node are the same, a cache miss can be serviced by main memory in the local node. Otherwise, the miss is forwarded to the *remote* home node. If the home node has a clean copy, it returns the block to the requesting cache. (We call this a 2-hop miss since it requires two network traversals — one from the requesting node to the home node, and the other back.) Otherwise, the read request is forwarded to the node that has the dirty copy. This node returns the block to the requesting node and also writes back the block to the home node. (We call this a 3-hop miss, as it takes three network traversals before the data is returned to the requesting node.)

If the block is not exclusively owned by a processor that issues a write request, a read-exclusive request is sent to the home node. The home node returns ownership and multicasts invalidation requests to any other nodes that have a copy of the block. Acknowledgments are returned directly to the issuing node so as to indicate when the write operation has completed. A write request to a dirty block is forwarded (the same way as a read request) to the node containing the dirty copy, which then returns the data.

The Stanford DASH multiprocessor [12] is an example of a CC-NUMA machine. The prototype is to consist of 16 processing nodes, each with 4 processors, for a total of 64 processors. Each processor has a 64 Kbytes first-level and a 256 Kbytes second-level cache. The interconnection network is a wormhole routed 2-D mesh network. The memory access latencies for a cache hit, local memory access, 2-hop, and 3-hop read misses are approximately 1, 30, 100, and 135 processor clocks respectively.

The CC-NUMA software model allows processes to be attached to specific processors and for data to be allocated from any specific node's main memory. However, the granularity at which data can be moved between different node's main memories (transparently to the application) is page sized chunks. We note that the allocation and movement of data between nodes may be done explicitly via code written by the application programmer, or automatically by the operating system [3]. This is in contrast to the COMA machines where such migration and replication happens automatically at the granularity of cache blocks.

2.2 COMA Architecture

Like CC-NUMA, a COMA machine consists of a number of processing nodes connected by an interconnection network. Each processing node has a high-performance processor, a cache, and a portion of the global shared memory. The difference, however, is that the memory associated with each node is augmented to act as a large cache, denoted *attraction memory* (AM) using DDM terminology [10]. Consistency among cache blocks in the AMs is maintained using a write-invalidate protocol. The AMs allow transparent migration and replication of data items to nodes where they are referenced.

In a COMA machine, the AMs constitute the only memory in the system (other than the disk subsystem). A consequence is that the location of a memory block is totally decoupled from its physical address. This creates several problems. First, when a reference misses in the local AM, a mechanism is needed to trace a copy of that block in some other node's AM. Unlike CC-NUMA, there is no notion of a home node for a block. Second, some mechanism is needed to ensure that the last copy of a block (possibly the only valid copy) is not purged.

To address the above problems and to maintain cache and memory consistency, COMA machines use a *hierarchical directory scheme* and a corresponding hierarchical interconnection network (at least logically so[1]). Each directory maintains state information about all blocks stored in the subsystem below. The state of a block is either exclusive in exactly one node or shared in several nodes. Note that directories only contain state information to reduce memory overhead; data are not stored.

Upon a read miss, a read request locates the closest node that has a copy of that block by propagating up the hierarchy until a copy in state shared or exclusive is found. At that point, it propagates down the hierarchy to a node that has the copy. The node returns the block along the same path as the request. (A directory read-modify-write needs to be done at each intermediate directory along the path, both in the forward and return directions.) Because of the hierarchical directory structure, COMA machines can exploit *combining* [7]; if a directory receives a read request to a block that is already being fetched, it does not have to send the new request up the hierarchy. When the reply comes back, both requesters are supplied the data.

A write request to an unowned block propagates up the hierarchy until a directory indicates that the copy is exclusive. This directory, the root directory, multicasts invalidation requests to all subsystems having a copy of the block and returns an acknowledgement to the issuing processor.

As stated earlier, decoupling the home location of a block from its address raises the issue of replacement in the AMs. A shared block (i.e., one with multiple copies) that is being replaced is not so difficult to handle. The system simply has to realize that there exist other copies by going up the hierarchy. Handling an exclusive block (the only copy of a block, whether in clean or dirty state) is, however, more complex since it must be transferred to another attraction memory. This is done by letting it propagate up in the hierarchy until a directory finds an empty or a shared block in its subsystem that can host the block.

Examples of COMA machines include the Swedish Institute of Computer Science's DDM machine [10] and Kendall Square Research's KSR1 machine [4]. The processing nodes in DDM are also clusters with multiple processors, as in DASH. However, the interconnect is a hierarchy of buses, in contrast to the wormhole-routed grid in DASH. In KSR1, each processing node consists of only a single processor. The interconnect consists of a hierarchy of slotted ring networks.

In summary, by allowing individual memory blocks to be migrated and replicated in attraction memories, COMA machines have the potential of reducing the number of cache misses that need to be serviced remotely. However, because of the hierarchy in COMA, latency for remote misses is usually higher (except when combining is successful); this may offset the advantages of the higher hit rates. We study these tradeoffs qualitatively in the next section.

[1]Wallach and Dally are investigating a COMA implementation based on a hierarchy embedded in a 3-dimensional mesh network [16].

3 Qualitative Comparison

In this section, we qualitatively evaluate the advantages and disadvantages of the CC-NUMA and COMA models. In particular, we focus on application data access patterns that are expected to cause one model to perform better than the other. We show that the critical parameters are the relative magnitudes of different miss types in the cache, and the spatial granularity of access to shared data. We begin with a discussion of the types of misses observed in shared-memory parallel programs, and discuss the expected miss latencies for CC-NUMA and COMA machines.

3.1 Miss Types and Expected Latencies

Since both CC-NUMA and COMA have private coherent caches, presumably of the same size, the differences in performance stem primarily because of differences in the miss latencies. In shared-memory multiprocessors that use a write-invalidate cache coherence protocol, cache misses may be classified into four types: cold misses, capacity misses, conflict misses, and coherence misses.

A *cold miss* is the result of a block being accessed by the processor for the first time. A *capacity miss* is a miss due to the finite size of the processor cache and a *conflict miss* is due to the limited associativity of the cache. For our discussion below, we do not distinguish between conflict and capacity misses since CC-NUMA and COMA respond to them in the same way. We collectively refer to them as capacity misses. A *coherence miss* (or an invalidation miss) is a miss to a block that has been referenced before, but has been written by another processor since the last time it was referenced by this processor. Coherence misses include both *false sharing misses* as well as *true sharing misses* [15]. False sharing misses result from write references to data that are not shared but happen to reside in the same cache line. True sharing misses are coherence misses that would still exist even if the block size were one access unit. They represent true communication between the multiple processes in the application.

We now investigate how the two models respond to cache misses of different types. Beginning with cold misses, we expect the average miss penalty to be higher for COMA, assuming that data is distributed among the main memory modules in the same way. The reason is simply that cold misses that are not serviced locally have to traverse the directory and network hierarchy in COMA. The only reason for a shorter latency for COMA would be if combining worked particularly well for an application.

For coherence misses, we again expect COMA to have higher miss latencies as compared to CC-NUMA. The reason is that the data is guaranteed not to be in the local attraction memory for COMA, and therefore it will need to traverse the directory hierarchy.[2] In contrast to CC-NUMA, the latency for COMA can, of course, be shortened if combining is successful, or if the communication is localized in the hierarchy.

Finally, for capacity misses, we expect to see shorter miss latencies for COMA. Most such misses are expected to hit in the local attraction memory since it is extremely large and

[2]We assume one processor per node. If several processors share the same AM (or local memory), a coherence miss can sometimes be serviced locally.

organized as a cache. In contrast, in CC-NUMA, unless data referenced by a processor are carefully allocated to local main memory, there is a high likelihood that a capacity miss will have to be serviced by a remote node.

In summary, since there are some kinds of misses that are serviced with lower latency by COMA and others that are serviced with lower latency by CC-NUMA, the relative performance of an application on COMA or CC-NUMA will depend on what kinds of misses dominate.

3.2 Application Performance

In this subsection, we classify applications based on their data access patterns, and the resulting cache miss behavior, to evaluate the relative advantages and disadvantages of COMA and CC-NUMA. A summary of this classification is presented in Figure 1. As illustrations, we use many applications that we evaluate experimentally in Section 5.

On the left of the tree in Figure 1, we group all applications that exhibit low cache miss-rates. Linear algebra applications that can be blocked, for example, fall into this category [11]. Other applications where computation grows at a much faster rate than the data set size, e.g., the $O(N^2)$ algorithm used to compute interaction between molecules in the Water application [14], often also fall into this category. In such cases, since the data sets are quite small, capacity misses are few and the miss penalty has only a small impact on performance. Overall, for these applications that exhibit a low miss rate, CC-NUMA and COMA should perform about the same.

Looking at the right portion of the tree, for applications that exhibit moderate to high miss rates, we differentiate between applications where coherence misses dominate and those where capacity misses dominate. Focusing first on the former, we note that this class of applications is not that unusual. High coherence misses can arise because an application programmer (or compiler) may not have done a very good job of scheduling tasks or partitioning data, or because the cache line size is too large causing false sharing, or because solving the problem actually requires such communication to happen. Usually, all three factors are involved to varying degrees. In all of these cases, CC-NUMA is expected to do better than COMA because the remote misses take a shorter time to service. As we will show in Section 5, even when very small processor caches are used (thus increasing the magnitude of capacity misses), at least half of the applications in our study fall into this category.

The situation where COMA has a potential performance advantage is when the majority of cache misses are capacity misses. Almost all such misses get serviced by the local attraction memory in COMA, in contrast to CC-NUMA where they may have to go to a remote node. CC-NUMA can deliver good performance only if a majority of the capacity misses are serviced by local main memory.

We believe that it is possible for CC-NUMA machines to get high hit rates in local memory for many applications that access data in a "coarse-grained" manner. By coarse grained, we mean applications where large chunks of data (greater than page size) are primarily accessed by one process in the program for significant periods of time. For example, many scientific applications where large data arrays are statically partitioned among the processes fall into this class. A specific example is the Cholesky sparse factorization algorithm that we evaluate later in the paper. In the Cholesky application, contiguous columns with similar non-zero structure (called supernodes) are assigned to various processors. If the placement is done right for such applications, the local memory hit rates can be very high.

Even if the locus of accesses to these large data chunks changes from one process to another over time, automatic replication and migration algorithms implemented in the operating system (possibly with hardware support) can ensure high hit rates in local memory for CC-NUMA. In fact, we believe that an interesting way to think of a CC-NUMA machine is as a COMA machine where the line size for the main-memory cache is the page size, which is not unreasonable since main memory is very large, and where this main-memory cache is managed in software. The latter is also not an unreasonable policy decision because the very-large line size helps hide the overheads.

In applications where many small objects that are collocated on a page are accessed in an interleaved manner by processes, page-level placement/migration obviously can not ensure high local hit-rates. An example is the Barnes-Hut application for simulating the interaction of N-body problems (discussed in Section 5). In this application multiple bodies that are accessed by several processors reside on a single page and, as a result, page placement does not help CC-NUMA.

In summary, we expect the relative performance of CC-NUMA and COMA to be similar if the misses are few. When the miss rate is high, we expect CC-NUMA to perform better than COMA if coherence misses dominate; CC-NUMA to perform similar to COMA if the capacity misses dominate but the data usage is coarse grained; and finally, COMA to perform better than CC-NUMA if capacity misses dominate and data usage is fine grained.

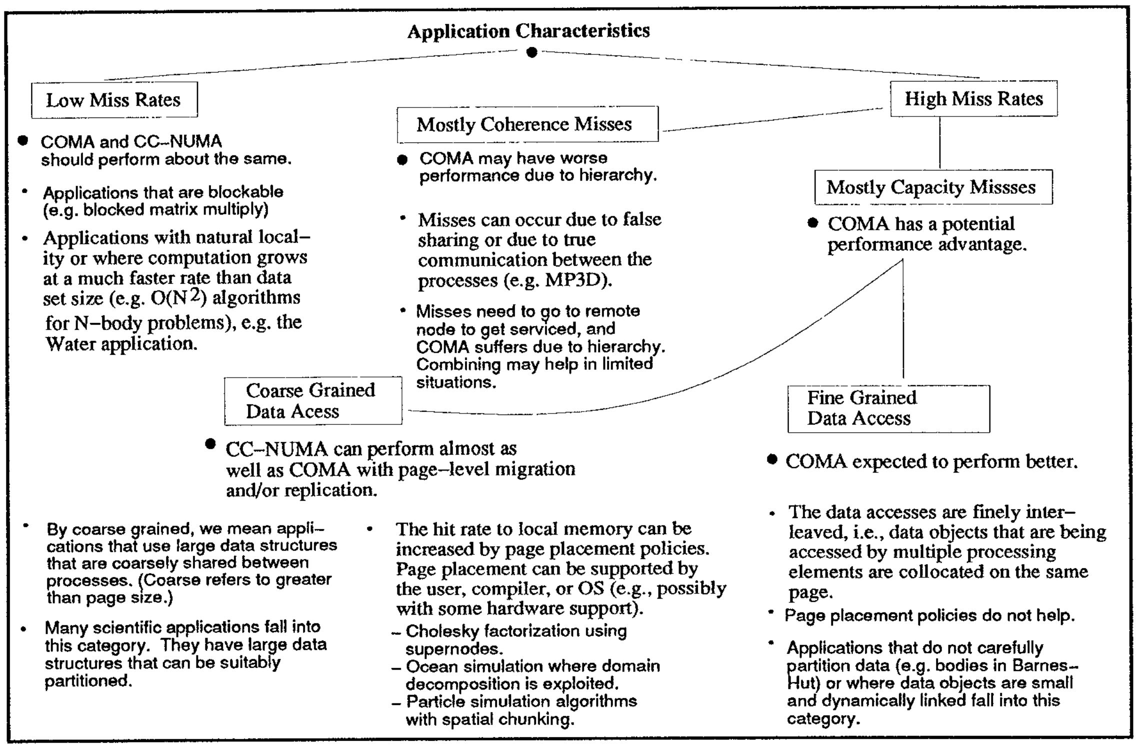
4 Experimental Methodology

This section presents the simulation environment, the architectural models, and the benchmark applications we use to make a quantitative comparison between CC-NUMA and COMA.

4.1 Simulation Environment

We use a simulated multiprocessor environment to study the behavior of applications under CC-NUMA and COMA. The simulation environment consists of two parts: (i) a functional simulator that executes the parallel applications and (ii) the two architectural simulators.

The functional simulator is based on Tango [5]. The Tango system takes a parallel application program and interleaves the execution of its processes on a uniprocessor to simulate a multiprocessor. This is achieved by associating a virtual timer with each process of the application and by always running the process with the lowest virtual time first. By letting the architectural simulator update the virtual timers according to the access time of each memory reference, a correct interleaving of all memory references is maintained. The architectural simulator takes care of references to shared data; instruction fetches and private data references are assumed to always hit in the processor cache.


Application Characteristics

Low Miss Rates

- COMA and CC–NUMA should perform about the same.
- Applications that are blockable (e.g. blocked matrix multiply)
- Applications with natural locality or where computation grows at a much faster rate than data set size (e.g. $O(N^2)$ algorithms for N-body problems), e.g. the Water application.

Mostly Coherence Misses

- COMA may have worse performance due to hierarchy.
- Misses can occur due to false sharing or due to true communication between the processes (e.g. MP3D).
- Misses need to go to remote node to get serviced, and COMA suffers due to hierarchy. Combining may help in limited situations.

High Miss Rates

Mostly Capacity Missses

- COMA has a potential performance advantage.

Coarse Grained Data Acess

- CC–NUMA can perform almost as well as COMA with page–level migration and/or replication.
- By coarse grained, we mean applications that use large data structures that are coarsely shared between processes. (Coarse refers to greater than page size.)
- Many scientific applications fall into this category. They have large data structures that can be suitably partitioned.
- The hit rate to local memory can be increased by page placement policies. Page placement can be supported by the user, compiler, or OS (e.g., possibly with some hardware support).
  - Cholesky factorization using supernodes.
  - Ocean simulation where domain decomposition is exploited.
  - Particle simulation algorithms with spatial chunking.

Fine Grained Data Access

- COMA expected to perform better.
- The data accesses are finely interleaved, i.e., data objects that are being accessed by multiple processing elements are collocated on the same page.
- Page placement policies do not help.
- Applications that do not carefully partition data (e.g. bodies in Barnes–Hut) or where data objects are small and dynamically linked fall into this category.

Figure 1: Prediction of relative performance between CC-NUMA and COMA based on relative frequency of cache miss types and application data access pattern.

4.2 Architecture Simulators

Both architectures consist of a number of processing nodes connected via low-latency networks. In our simulations we assume a 16 processor configuration, with one processor per processing node. (Figure 2(a) shows the organization of the processing node.) We assume a cache line size of 16 bytes. In the default configurations, we use a processor cache size of 4 Kbytes, and in the case of COMA, infinite attraction memories. For COMA we assume a branching factor of 4, implying a two-level hierarchy.

The reasons for choosing this rather small default processor cache size are several. First, since the simulations are quite slow, the data sets used by our applications are smaller than what we may use on a real machine. As a result, if we were to use full-size caches (say 256 Kbytes), then for some of the applications all of the data would fit into the caches and we would not get any capacity misses. This would take away all of the advantages of the COMA model, and the results would obviously not be interesting. Second, by using very small cache sizes, we favor the COMA model, and our goal is to see whether there are situations where CC-NUMA can still do better. Third, 4 Kbytes is only the default, and we also present results for larger cache sizes. As for use of infinite attraction memories, our choice was motivated by the observation that the capacity miss-rates are expected to be extremely small for the attraction memories. As a result, the complexity of modeling finite sized attraction memories did not seem justified.

We now focus on the latency of cache misses in CC-NUMA and COMA. To do this in a consistent manner, we have defined a common set of primitive operations that are used to construct protocol transactions for both architectures. We can then choose latency values for these operations and use them to evaluate the latency of memory operations. In the following subsections, we describe reference latencies for the two architectures in terms of these primitive operations. Table 1, located at the end of this section, summarizes these primitive operations and lists the default latency numbers, assuming a processor clock rate of 100 MHz.

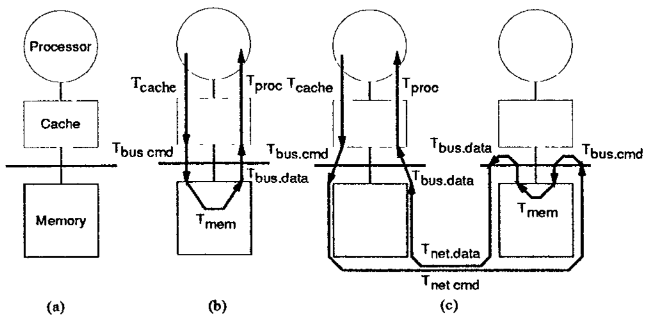


Figure 2: Node organization (a) and latency calculations for local and remote misses for CC-NUMA (b) and (c).

4.2.1 CC-NUMA Latencies

For CC-NUMA, a load request that hits in the cache incurs a latency of a cache access, T_{cache}. For misses, the processor is stalled for the full service latency of the load request.

In Figure 2(b) we depict the memory latency for a load

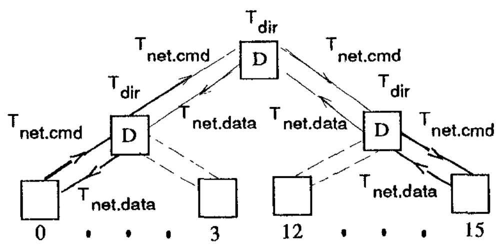


Figure 3: Latency model for global read requests in COMA.

request that hits in the local memory. The load first checks the cache (T_{cache}), it then arbitrates for the local bus and places the request onto the bus ($T_{bus.cmd}$), it then accesses memory (T_{mem}), the data is placed on the local bus ($T_{bus\,data}$), and finally the cache loads data from the bus and restarts the processor (T_{proc}).[3] Similarly, in Figure 2(c) we show the latency for a load request to a shared block whose home is not the local node. $T_{net\,cmd}$ is the network latency to send the load request to the home node, and $T_{net.data}$ is the network latency for the return data packet. If the block is dirty in a node other than the home node, an additional latency that consists of another network traversal and node memory access is needed.

Stores are handled according to the weakly ordered consistency model [6] by a write buffer which sits between the processor and the cache. Hence, the processor is not stalled on a store request. However, on a synchronization request, the processor is stalled until all pending invalidation acknowledgements have been received.

The default network latency numbers assume a 4×4 wormhole-routed synchronous mesh clocked at 100 MHz with 16-bit wide links.

4.2.2 COMA Latencies

For COMA, the processor cache hit latency is the same as CC-NUMA. The latency for a hit in the attraction memory is the same as a request serviced by local memory in CC-NUMA. For requests that need to go to a remote processing node, we illustrate the latency using a simple example. Figure 3 shows a request made by processor 0 and serviced by processor 15. The latency consists of a check in processor cache (T_{cache}); request issue on local bus ($T_{bus.cmd}$); check in local attraction memory (T_{mem}); traversal up through the hierarchy ($T_{net.cmd} + T_{dir}$ at each level); traversal down the hierarchy ($T_{net.cmd} + T_{dir}$ at each level); lookup in the memory of processor 15 ($T_{bus.cmd} + T_{mem} + T_{bus.data}$); traversal back up the hierarchy ($T_{net\,data} + T_{dir}$ at each level); traversal down to the requesting node ($T_{net.data} + T_{dir}$ at each level); and finally back to the processor ($T_{bus.data} + T_{proc}$). Latency for other requests can be similarly derived. The protocol follows that used by DDM [10], assuming infinite write buffers and a weakly ordered consistency model. The effects of request combining in the hierarchy are also modeled in our simulator. However, we do not model contention at any of the buses or directories.

A hierarchical network with point-to-point links with a width of 32-bits and synchronous data transfers is assumed. Since a hierarchical layout makes it difficult to achieve high clock rates (unlike a mesh), we assume that the links are clocked at 50 MHz yielding the latency numbers $T_{net.cmd} = 4$ and $T_{net\,data} = 12$ pclocks, respectively.

Table 1: Default latencies for primitive operations in processor clock cycles (1 pclock $\approx$ 10 ns).

Primitive Operation	Parameter	Latency (pclocks)
Cache Access Time	T_{cache}	1
Cache Fill and Restart	T_{proc}	6
Local Bus Request Time	$T_{bus.cmd}$	4
Local Bus Reply Time	$T_{bus.data}$	2
Com. Net. Latency (CC-NUMA)	$T_{net.cmd}$	12
Data Net. Latency (CC-NUMA)	$T_{net.data}$	20
Com. Net. Latency (COMA)	$T_{net.cmd}$	4
Data Net. Latency (COMA)	$T_{net\,data}$	12
Memory and AM	T_{mem}	20
Directory Update (rd-mod-wr)	T_{dir}	20

Table 2: Default latencies for various read operations.

Read Operation	Latency (pclocks)
Cache Hit	1
Fill from Local Node	33
2-hop Remote Fill (CC-NUMA)	71
3-hop Remote Fill (CC-NUMA)	109
1-level Remote Fill (COMA)	131
2-level Remote Fill (COMA)	243

In Table 2, we show the default latencies for read requests satisfied at various levels in the memory hierarchy, based on the default latencies for the primitive operations from Table 1. Note that these are just default latencies. We will present results for other architectural assumptions as well in Section 5.6.

4.3 Benchmark Programs

To understand the relative performance benefits of CC-NUMA and COMA we use a variety of scientific and engineering applications, including all six SPLASH benchmarks [14]. A summary of the eight applications that we use is given in Table 3.

The data sets used for our eight applications are as follows. MP3D was run with 10K particles for 10 time steps. PTHOR was run with the RISC circuit for 5000 time steps with incremental deadlock detection turned on. LocusRoute used the circuit Primary1.grin which has 1266 wires and a 481x18 cost array. Water was run with 288 molecules for 4 time steps. Cholesky was run using the matrix bcsstk14 from the Boeing-Harwell benchmark matrices, and it has 1806 equations and 61648 non-zeroes. LU was run on a 200×200 random matrix. Barnes-Hut [13] was run using 2048 bodies in a plummer distribution simulated for 10 time steps with a tolerance of 1.0. Finally, Ocean was run with a 98×98 grid with a convergence tolerance of 10^{-7} and w set to 1.15. Statistics acquisition is

[3]Note: The latency for $T_{bus\,data}$ is only one bus cycle (2 pclocks) in Figure 2(b) because the return of data overlaps with the latency for processor restart T_{proc}.

Table 3: Benchmark programs.

Benchmark	Description
MP3D	Particle-based wind tunnel simulation
PTHOR	Distributed-time logic simulation
LocusRoute	VLSI standard cell router
Water	Molecular dynamics code: Water
Cholesky	Cholesky factorization of sparse matrix
LU	LU decomposition of dense matrix
Barnes-Hut	N-body problem solver O(NlogN)
Ocean	Ocean basin simulation

started when the parallel section of the application is entered (see SPLASH report [14]), because the initialization part is expected to be negligible for full-scale runs.

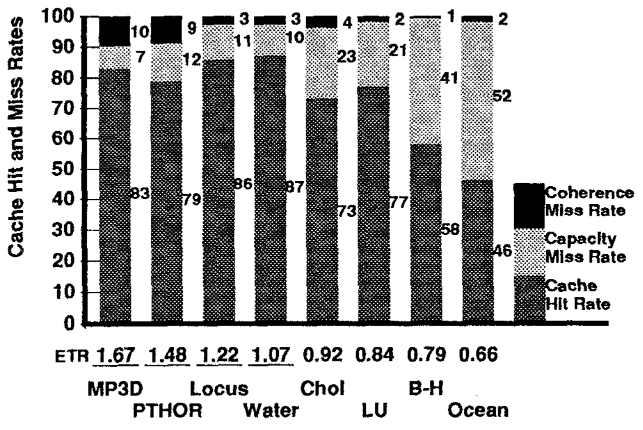


Figure 4: Cache hit-rate, capacity miss-rate, and coherence miss-rate assuming 4 Kbyte caches. ETR is the execution time for COMA divided by the execution time for CC-NUMA.

5 Quantitative Evaluation

In this section, we provide experimental results that show the advantages and disadvantages of CC-NUMA and COMA. We begin by investigating how the relative frequency of capacity and coherence misses impacts the relative performance of CC-NUMA and COMA. Then in Sections 5.2, 5.3, and 5.4, we explore how page migration and initial page placement can help CC-NUMA reduce the penalty of capacity misses, thus improving its performance. Finally, in Sections 5.5 and 5.6, we study how our results are affected by variations in architectural parameters.

5.1 Performance of CC-NUMA and COMA

Since performance of CC-NUMA and COMA machines is closely tied to the cache hit-rate achieved and the types of cache misses incurred, in Figure 4 we present relevant statistics. For each application, the bottom dark-gray section gives the cache hit-rate, the middle light-gray section gives the capacity miss-rate, and the top black section gives the coherence miss-rate. We do not show cold miss-rates separately because they are very small (0.9% for Cholesky, 0.26% for LocusRoute, and less than 0.1% for the remaining applications), and in Figure 4 they are lumped with capacity misses.

As can be seen from Figure 4, there is a large variation in the relative magnitude of capacity and coherence misses across the applications; for example, while coherence misses dominate in MP3D, capacity misses dominate in Ocean. To see how this impacts the relative performance of CC-NUMA and COMA, we also measured the *node hit-rate* for the applications, that is, the fraction of references that get serviced by the cache or the local memory. Although we do not present these data here directly, for COMA machines, the node hit-rate is essentially the cache hit-rate plus the capacity miss-rate, since all capacity misses are serviced by the local attraction memory. In contrast, for CC-NUMA, the node hit-rate is highly dependent on the way in which the data are distributed among the processing nodes. If we assume that the data pages are distributed randomly or in a round-robin manner (as default in this paper, we use the round-robin strategy), then node hit-rate for CC-NUMA is expected to be the cache hit-rate plus the capacity miss-rate divided by the number of processors. Given that the number of processors is quite large, 16 in our case, the node hit-rate for CC-NUMA is approximately the same as the cache hit-rate. Thus the difference in the node hit-rate between COMA and CC-NUMA is roughly the middle light-gray section in Figure 4.

The question now becomes how the differences in node hit-rate and node miss penalty for CC-NUMA and COMA impact their relative performance. We use *execution time ratio* (ETR) of COMA to CC-NUMA as a measure of the relative performance. Thus ETR > 1 implies that CC-NUMA is performing better than COMA. In Figure 4, we show the ETR beneath the hit/miss rate bar for each application. As expected, for applications where the coherence miss-rate is a significant part of the overall miss-rate (MP3D, PTHOR, LocusRoute, and Water), COMA exhibits worse performance than CC-NUMA due to the higher node miss penalty incurred by COMA. For the other four applications (Cholesky, LU, Barnes-Hut, and Ocean), the capacity miss-rate dominates the overall miss-rate, and as expected, COMA shows better performance. We also observe that the overall swing in the relative performance is quite large; while CC-NUMA does 67% better than COMA for MP3D, COMA does 52% better than CC-NUMA for Ocean.

To study whether combining is playing an important role in the higher performance of COMA, we measured the percentage of node misses that get combined in the hierarchical directory structure. Combining turned out to be significant in one case only — in LU, 47% of all remote requests get combined. This is because the pivot column is read by all processors as soon as all modifications to it have been completed. Except for the LU application, combining was of little help in reducing the node miss penalty as incurred by the hierarchical directory structure in COMA — in Barnes-Hut, 6% of remote requests get combined, and for all remaining applications, less than 1% of remote requests get combined.

To summarize, we have shown that the relative frequency of capacity and coherence misses has a first order effect on the relative performance of CC-NUMA and COMA. We observed that four of the eight applications perform worse on COMA because of the coherence miss penalty associated with the hierarchical network. We also observed that request combining

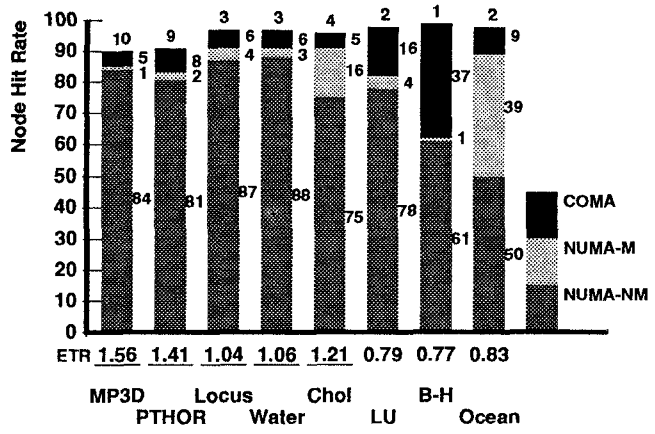


Figure 5: Node hit-rate for CC-NUMA with no migration (NUMA-NM), CC-NUMA with migration (NUMA-M), and COMA assuming 4 Kbytes pages. ETR is the execution time for COMA divided by the execution for NUMA-M.

has only a small impact in hiding the higher network latency incurred by the hierarchy in COMA.

5.2 CC-NUMA: Page Migration

As stated in Section 3, if large chunks of data (greater than page size) are primarily accessed by one process in the program for significant periods of time, intelligent page migration and replication algorithms can help CC-NUMA reduce the penalty for capacity misses. In this subsection, we focus on the performance gains for CC-NUMA if the operating systems performed page migration. We consider a fairly aggressive competitive page migration algorithm proposed by Black et al. [2]. Note, our purpose here is to explore the performance potential of page migration rather than advocating the use of this particular page migration algorithm.

Our migration algorithm associates N counters with each page, given N processing nodes. A remote access to a page increments the corresponding counter and decrements some other randomly chosen counter. When a counter exceeds twice the migration cost (measured in units of remote access cost), the page is migrated to the corresponding node. We have estimated the software overhead of migration by examining the operating system code associated with page transfers. It includes invocation of the page management software (800 pclocks); invalidation of all TLB entries for the page ($300+40n$ pclocks, assuming n processors have TLB entries); and finally, migration of the page ($T_{mem}P/B$, where P is the page size and $B = 16$ is the block size). We also assume that the page is blocked from being accessed while it is being moved.

To see how well page migration manages to improve the node hit-rate for CC-NUMA, in Figure 5 we show the node hit-rate for CC-NUMA with no page migration (NUMA-NM), CC-NUMA with page migration (NUMA-M), and COMA assuming 4 Kbyte pages. The number on top of each bar represents the coherence miss-rate. We see that the node hit-rate improves significantly for Cholesky and Ocean for NUMA-M, since data usage in both applications is coarse grained. The Cholesky application works with groups of contiguous columns, called supernodes, that are often larger than a page. The Ocean application works with arrays of data objects that are coarsely partitioned and assigned to each processor in chunks which are typically larger than 4 Kbytes.

Table 4: Node hit-rate and execution time for NUMA-M (with different page sizes) relative to NUMA-NM (with 4Kbyte pages).

Page Size	LU Node HR	LU Rel. Exec.	LU Mig. Count	Ocean Node HR	Ocean Rel. Exec.	Ocean Mig. Count
NUMA-NM	78	1.00	–	50	1.00	–
NUMA-M 4K	82	1.06	2.7K	89	0.79	6.6K
NUMA-M 2K	86	1.02	4.6K	93	0.68	6.6K
NUMA-M 1K	89	0.97	3.8K	96	0.64	4.7K
NUMA-M .5K	90	0.93	4.0K	96	0.66	7.8K
COMA	98	0.84	–	98	0.66	–

Finally, to show the overall benefits of page migration, we present the execution time ratio (ETR) of COMA to NUMA-M beneath each bar in Figure 5. The four applications where previously NUMA-NM did better than COMA (MP3D, PTHOR, LocusRoute, and Water), NUMA-M continues to do better than COMA. The performance advantage is, however, slightly smaller now — in these applications the software overhead of migration is slightly larger than the benefits. However, for Cholesky, where previously NUMA-NM did worse than COMA (see Figure 4), NUMA-M does significantly better than COMA (ETR = 1.21). For Ocean, NUMA-M substantially decreases the performance advantage of COMA than before. For LU and Barnes-Hut the performance is essentially unchanged between NUMA-NM and NUMA-M. We speculate that one reason why the gains are small is that we have been using small data sets for our applications. To study the effects of larger data sets, we consider page size variations next.

5.3 Page Migration: Impact of Page Size

As stated in Section 4, the data set size we use for the applications is smaller than what we may use on real machines. Consequently, for applications that use coarse data partitioning, we expect data chunks to be larger for full-sized problems. For example, as matrix size is increased for LU, the columns will get larger. In this section, we indirectly study the performance gains from migration for larger data sets by instead considering smaller page sizes (512b, 1K, and 2K pages). Due to space limitations, we only present results for LU and Ocean. These two are among the three applications that are currently doing worse under COMA; the results for Barnes-Hut, the third application, are not expected to change with page size.

Table 4 shows the node hit-rate and the execution time relative to NUMA-NM (4 Kbyte pages) for various page sizes for LU and Ocean. We see that the node hit-rate increases as the page size is decreased. The explanation is that as the page size is decreased, the data partitions in these applications start becoming larger than the pages. As a result, pages are primarily referenced by only a single processor, and quickly migrate to that processor. In contrast, with larger pages there is false sharing with multiple processors referencing a page, and the migration algorithm can not satisfy all referencing processors. Responding to increasing node hit-rates, execution times go down as the page size is reduced. For 512 byte pages, the difference between the performance of NUMA-M and COMA is less than 10% for LU and none for Ocean.

For Ocean, it is interesting to note that the minimum execution time occurs for 1 Kbyte pages and not for 512 byte pages. The reason is simply the overhead due to page migrations. To understand this, we also show the number of migrations that occur for a given page size in Table 4. For Ocean, the number of migrations follows a U-shaped curve. For page sizes much larger than data objects, we have many migrations because of false sharing. For page sizes much smaller than data objects, we have many migrations because bringing each object closer takes multiple page migrations. For LU, it is interesting to note that the node hit-rates for NUMA-M never reach close to that achieved by COMA. The reason is that columns in the matrix do not occupy an integer number of pages. Since successive columns are assigned to different processors, the migration algorithm never succeeds in totally satisfying all processors. In contrast, COMA with its 32-byte memory blocks does very well in bringing data close to the processors.

To summarize, we see that for the scaled down problems used in this paper, migration with smaller page sizes is quite effective. In turn, the results also indicate that for full-sized problems, we are likely to be able to get good performance while using migration with regular sized pages. Of course, page migration is not expected to work for all applications. For applications such as Barnes-Hut, where data chunks are much smaller than the page size, the migration overheads are likely to exceed the benefits.

5.4 CC-NUMA: Benefits from Initial Placement

A disadvantage of using page migration algorithms is that they can require substantial hardware and software support. An alternative is to let the compiler/programmer spend some effort in partitioning the data set and in placing the pages so as to improve the local memory hit-rate. To study the benefits of initial placement, without doing the placement ourselves, we adopted the following scheme. We evaluated the performance of the applications using the same page migration algorithm as in the last subsection, but with only a single migration allowed for each page. In Table 5, we show the node hit-rate, the execution time ratio, and the number of migrations for LU and Ocean under this scheme (NUMA-I) with various page sizes. (The reasons for considering only LU and Ocean are the same as in the previous subsection.)

First, we note that the node hit-rate increases as the page size is reduced. This is what we expect since false sharing of pages is reduced. Second, execution time also drops as the page size goes down. Third, looking at the number of migrations, the general trend is that the migration count goes up as page size is decreased. The reason is that twice as many pages must be migrated when the page size is reduced by a factor of two.

Comparing the relative performance of NUMA-I and NUMA-M, we see that LU does significantly better with single migrations than with multiple migrations (see Table 4), in fact, even better than COMA with 512 byte pages. The reason is that we have significantly reduced the number of page migrations (e.g., down from 2.7K to 135 migrations for 4 Kbyte pages), thus reducing the software overhead of migration. When multiple migrations are allowed, a page in LU may thrash back-and-forth between the various processors that have columns allocated on that page, without really improving the overall hit-rate in local memory.

For Ocean, the performance for single versus multiple page migrations is essentially the same. Although the node hit-rate is lower when only a single migration per page is allowed, this is compensated by the fact that there are fewer page migrations and hence lower software overhead.

Table 5: Node hit-rate and execution time for CC-NUMA with single page migration (NUMA-I) relative to NUMA-NM for various page sizes.

Model	LU			Ocean		
	Node HR	Rel. Exec	Mig. Count	Node HR	Rel. Exec	Mig. Count
NUMA-NM	78	1.00	–	50	1.00	–
NUMA-I 4K	83	0.96	135	87	0.74	581
NUMA-I 2K	87	0.88	270	89	0.66	1091
NUMA-I 1K	90	0.87	511	95	0.62	2218
NUMA-I .5K	91	0.79	948	94	0.64	4250
COMA	98	0.84	–	98	0.66	–

In summary, we show that initial placement manages to eliminate most performance differences between CC-NUMA and COMA for LU and Ocean. (All other applications, with the exception of Barnes-Hut, already do quite well with CC-NUMA.) Of course, proper intial placement requires extra effort from the compiler/programmer, and this must be traded off against the higher implementation complexity of COMA. COMA, however, is expected to be more responsive to dynamically changing workloads (e.g., multiprogrammed workloads).

5.5 Impact of Cache Size Variations

All experiments so far have been based on 4 Kbyte caches. In this subsection we study how the relative frequency of capacity and coherence misses changes as the cache size is increased, while keeping the problem size fixed.

An important effect of increasing the cache size is that while the capacity misses go down, the coherence misses remain the same. As a result, with larger caches, we expect coherence misses to start dominating, thus making the relative performance of CC-NUMA better than COMA. In Table 6 we show this data for the four applications (Cholesky, LU, Barnes-Hut, and Ocean) that did worse on CC-NUMA than COMA in Section 5.1.

As the cache size is increased, we clearly see that the ca-

Table 6: Capacity miss-rate (on left) and execution time ratio of COMA versus NUMA-NM (on right) for various cache sizes. The bottom line of the Table shows the coherence miss-rate for the applications.

Cache Size	Cholesky	LU	B-H	Ocean
4 K	23/0.92	21/0.84	41/0.79	52/0.66
16 K	6/1.52	8/1.08	20/0.89	28/0.79
64 K	3/1.58	3/1.35	5/1.01	15/0.91
Coher. MR	4	2	1	2

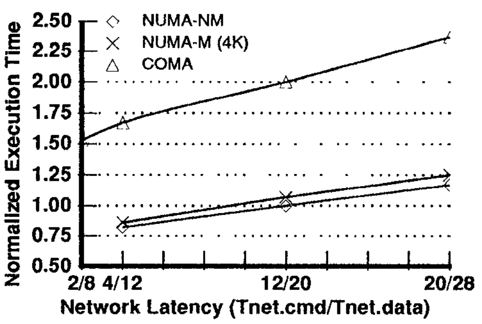


Figure 6: Network latency variation for MP3D.

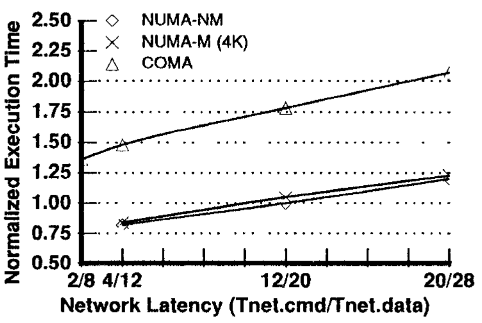


Figure 7: Network latency variation for PTHOR.

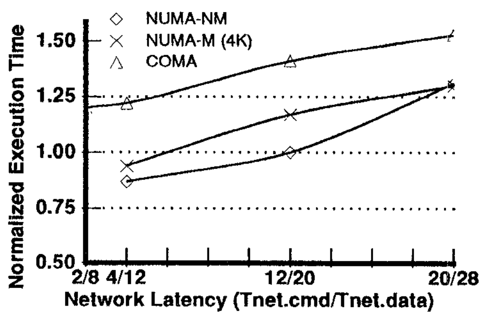


Figure 8: Network latency variation for LocusRoute.

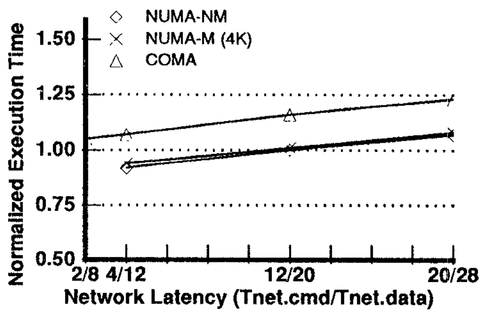


Figure 9: Network latency variation for Water.

pacity miss-rate is reduced and the coherence miss-rate starts to dominate. As expected, with larger caches the execution time ratio increases indicating that CC-NUMA is starting to perform better. As the data show, with 64 Kbyte caches, the performance of CC-NUMA (with no migration) is better for all applications except Ocean.

5.6 Impact of Network Latency Variations

We have seen that COMA's performance is limited primarily by the larger latency incurred by its hierarchical directory structure. In this subsection we explore how variations in the network latency and directory access times influence the relative performance. Due to space limitations, we only present results for MP3D, PTHOR, LocusRoute, and Water. By picking this set of applications that perform better on CC-NUMA, we wish to see if it is possible for COMA to perform better if a more aggressive network implementation is assumed. We will first study variations in network latency and then consider more aggressive directory implementations.

As default values for network latency, we have so far assumed a synchronous point-to-point hierarchy for COMA with latency numbers $T_{net.cmd} = 4$ and $T_{net\,data} = 12$ pclocks. For CC-NUMA, we have assumed a synchronous mesh with default latency numbers $T_{net.cmd} = 12$ and $T_{net.data} = 20$ pclocks. In Figures 6–9 we show the relative performance of COMA, NUMA-M (with 4 Kbyte pages), and NUMA-NM under different network latency assumptions. All execution times are shown relative to NUMA-NM with default parameters. An important observation is that even if we consider a very aggressive point-to-point hierarchy for COMA ($T_{net.cmd} = 2$ and $T_{net\,data} = 8$), default NUMA-NM outperforms COMA for MP3D, PTHOR, LocusRoute, and Water. We study variations in directory access time for COMA next.

The directory access time is another important contributor to the node miss penalty associated with COMA. A directory access consists of a read-modify-write cycle in order to modify the state information. More specifically, the following basic actions and latencies are associated with a directory access, assuming 80ns DRAM-based directories and a 100 MHz processor clock rate: (i) arbitration among the multiple inputs into a directory (2 pclocks); (ii) directory DRAM access plus data buffering (10 pclocks); (iii) tag comparison and decision about next action (4 pclocks); (iv) directory DRAM update (6 pclocks); and (v) precharge before next directory access (8 pclocks). The busy time thus adds up to 30 pclocks. If the network load is low, the directory update and precharge operations can be overlapped by network transfers. We have partly taken this into account in our default directory access time by assuming $T_{dir} = 20$ pclocks. However, for applications with high coherence miss-rates, less overlap is expected. In Table 7 we show the execution time of COMA relative to CC-NUMA with default parameters, assuming less directory overlap ($T_{dir} = 28$ pclocks). We see that for applications with a significant coherence miss-rate, such as MP3D and PTHOR, less overlap can substantially degrade the performance of COMA.

One could reduce the node miss penalty for COMA by using faster (and more expensive) SRAM-based directories. In order to study the effect of faster directories, in Table 7 we show the performance of COMA (relative to CC-NUMA with default parameters), assuming directories built from 30ns SRAMs and where the update and precharge operations are completely overlapped by the network transfer ($T_{dir} = 12$ pclocks). We see that even for such an aggressive directory implementation, COMA performs worse than CC-NUMA for all four applications.

In summary, we note that even for more aggressive im-

Table 7: Execution time ratio of COMA to NUMA-NM as directory access time is varied.

T_{dir}	MP3D	PTHOR	Locus	Water
28	1.92	1.68	1.42	1.12
20	1.67	1.48	1.22	1.07
12	1.41	1.26	1.19	1.03

plementations of the hierarchical directory structure, COMA suffers from the higher network latency for applications with a significant coherence miss-rate.

6 COMA-F: A Flat COMA

We have seen that COMA's primary advantage is small capacity-miss penalties due to the large attraction memories, and its primary disadvantage is large coherence-miss penalties due to the hierarchical directory structure. The hierarchy is fundamental to COMA's coherence protocol because it helps locate copies of a memory block in the system. It also helps to support combining of requests. In contrast, CC-NUMA has an explicit home node for each memory block, and the directory at the home node keeps track of all copies of that memory block. Consequently, a copy of a memory block can be located without traversing any hierarchy, resulting in lower remote-miss latencies. In this section, we propose a new architecture called COMA-F (for COMA-FLAT) that provides the benefits of both COMA and CC-NUMA (low capacity-miss penalties and low coherence-miss penalties). To avoid confusion, we denote previously proposed COMA machines as COMA-H, for hierarchical COMA.

COMA-F has no hierarchy. However, like COMA-H, COMA-F supports migration and replication of cache blocks at the main-memory level by organizing its main memory as an attraction memory (i.e., there are tags associated with each memory block). In addition, like CC-NUMA, each memory block has a clear notion of a home node, and the directory memory there keeps track of all copies of that block. However, unlike CC-NUMA, the home node does not reserve its main memory for blocks for which it keeps directory entries (thus, blocks whose home is some other processing node can come and reside there). Below, we present an overview of the memory coherence protocol for COMA-F — the detailed protocol is described in [8].

We begin with the structure of the directory entries. Each entry contains a pointer to one of the copies denoted MASTER, and a list of other nodes having a copy of the block, called the sharing list. The state of a directory entry can either be SHARED or EXCLUSIVE. The state of an attraction-memory block can be one of INVALID, SHARED, MASTER-SHARED, or EXCLUSIVE. If a block is in MASTER-SHARED or EXCLUSIVE state, it implies that the directory entry considers the local node to be the MASTER. As for directory-memory overhead for COMA-F, we note that it should be possible to use limited-pointer directory schemes as proposed for CC-NUMA, but sparse directories are not expected to work well [9].

If a read request misses in the attraction memory, it is sent to the HOME node. The HOME forwards the read request to the MASTER node which responds with a copy of the data to the requesting node. The HOME also updates the sharing list. If the MASTER copy was in state EXCLUSIVE, the state of the directory entry is changed to SHARED and the block state in the MASTER node is changed to MASTER-SHARED. Note, in contrast to CC-NUMA, accessing a clean block in COMA-F may take three instead of two network traversals. Also in contrast to COMA-H, there is no hardware combining in COMA-F.

A read-exclusive request is also first sent to the HOME node. As before, the HOME node forwards the request to the MASTER node which responds with a copy of the block to the requesting node. In addition, the requesting node now becomes the new MASTER node and all other copies are invalidated. Acknowledgements are sent directly to the new MASTER node.

Since there is no physical memory backing up the attraction-memory cache blocks, replacements need special care in order to avoid replacing the last remaining copy. A node that initiates replacement of a SHARED block can simply discard it and then inform the HOME node to remove itself from the sharing list. However, if the state of the block is either EXCLUSIVE or MASTER-SHARED, then the initiating node is currently the MASTER node and it must send the replaced block to the HOME node. The HOME node must now nominate a new MASTER node. If there are other nodes with copies, an arbitrary node among them is nominated as the MASTER. If there are no other copies, then we need to get fresh space for the replaced block. While there are many different strategies possible, one reasonable strategy is to make a place for the replaced block in the HOME node itself. While this may cause another replacement, the chain of replacements is expected to be short.

We have investigated the performance of COMA-F using the protocol mentioned above, assuming infinite attraction memories. In Table 8, we present execution times for COMA-H and NUMA-NM relative to COMA-F, assuming 4 Kbyte pages and default latency numbers. As expected, COMA-F outperforms both COMA and NUMA-NM. Compared to COMA-H, COMA-F manages to reduce the coherence-miss penalty (as seen from performance of MP3D and PTHOR) and as compared to CC-NUMA, COMA-F manages to reduce the capacity-miss penalty (as seen from performance of LU, Barnes-Hut, and Ocean).

7 Conclusion

We have studied the relative performance of two different approaches to designing large-scale shared-memory multiprocessors, CC-NUMA and COMA. We have found that for applications with low miss rates, the two styles of machines achieve nearly identical performance. The performance differences arise as a result of the efficiency with which they handle cache misses. The COMA style handles capacity misses more efficiently than CC-NUMA. The replication and migration of small blocks of data at the main-memory level allows COMA to service a majority of such misses locally. The CC-NUMA style handles coherence misses more efficiently than COMA. The hierarchical directory structure that is needed by COMA results in a much larger latency than the flat structure in CC-NUMA. We have observed through simulation that the relative performance of these two machine models is easily predicted by the relative frequency of these two types of misses.

In contrast to COMA machines, CC-NUMA can replicate

Table 8: Execution time ratio of COMA-H and NUMA-NM relative to COMA-F.

	MP3D	PTHOR	Locus	Water	Chol	LU	B-H	Ocean
COMA-H	1.90	1.73	1.30	1.16	1.35	1.27	1.04	1.11
NUMA-NM	1.14	1.18	1.09	1.09	1.46	1.51	1.32	1.68

and migrate data at the main-memory level only in large page-sized chunks. However, we show that proper initial placement and smart migration of pages can still be quite effective for scientific applications that use coarse-grained data partitions. Overall, with page migration turned on, we show that CC-NUMA performs better than COMA or is competitive with COMA for seven of the eight benchmark applications that we evaluate.

Finally, by combining the notion of a home location for directory information with the feature of replication of cache-block sized chunks at the main-memory level, we have outlined a COMA-F architecture that does not rely on a hierarchical directory structure. Our preliminary results show that such an architecture can offer significant performance improvements. However, we believe it is still an open question whether the additional complexity and hardware overhead of COMA machines is justified by the expected performance gains; this question is especially critical for hierarchical COMA machines.

Acknowledgments

We would like to thank Ed Rothberg, Kourosh Gharachorloo, Rohit Chandra, John Hennessy, and Kunle Olukotun for discussions and helpful comments on earlier drafts of this paper. This work has been supported by DARPA contract N00039-91-C-0138. Per Stenström is supported by STU contract 9001797. Anoop Gupta is also supported by an NSF Presidential Young Investigator Award.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[2] David L. Black, Anoop Gupta, and Wolf-Dietrich Weber. Competitive management of distributed shared memory. In *Proceedings of Compcon 1989*, March 1989.

[3] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the 4th International Conerfernce on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, 1991.

[4] Henry Burkhardt III, Steven Frank, Bruce Knobe, and James Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[5] Helen Davis, Stephen R. Goldschmidt, and John L. Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of International Conference on Parallel Processing*, pages 99–107, 1991. Vol. II.

[6] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[7] Alan Gottlieb, Ralph Grishman, C. Kruskal, Kevin McAuliffe, Larry Rudolph, and Mark Snir. The NYU Ultracomputer - Designing a MIMD, shared memory parallel machine. *IEEE Transactions on Computers*, 32(2):175–189, February 1983.

[8] Anoop Gupta, Truman Joe, and Per Stenström. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. Technical report, Stanford University, March 1992.

[9] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of International Conference on Parallel Processing*, August 1990.

[10] Erik Hagersten, Seif Haridi, and David H.D. Warren. The cache-coherence protocol of the data diffusion machine. In Michel Dubois and Shreekant Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*. Kluwer Academic Publishers, 1990.

[11] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conerfernce on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[12] Daniel E. Lenoski, James P. Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. 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.

[13] Jaswinder P. Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. Load balancing and data locality in parallel hierarchial N-body simulation. Technical Report CSL-TR-92-505, Stanford University, February 1992.

[14] Jaswinder P. Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[15] Joseph Torrellas, Monica S. Lam, and John L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proceedings of the International Conference on Parallel Processing*, pages 266–270, 1990. Vol. II.

[16] Deborah A. Wallach. A scalable hierarchial cache coherence protocol. Bachelor of Science Thesis, Massachuesetts Institute of Technology, May 1990.