Cache-Only Memory Architectures

COMA reduces the impact of frequent long-latency memory accesses by automatically replicating and migrating data across memory modules.

Fredrik Dahlgren
Ericsson Mobile Communications

Josep Torrellas
University of Illinois at Urbana-Champaign

Scalable shared-memory multiprocessors are emerging as attractive platforms for applications with high-performance demands. What makes these machines attractive is the shared address space, which allows processors in a multiprocessor to share data the same way it is shared by multiple processes in a sequential machine. The shared-memory paradigm makes it easier to write parallel programs, but tuning the application to reduce the impact of frequent long-latency memory accesses still requires substantial programmer effort. Researchers have proposed using compilers, operating systems, and architectures to improve performance by placing data close to the processors that use it.

The Cache-Only Memory Architecture (COMA) increases the chances of data being available locally because the hardware transparently replicates the data and migrates it to the memory module of the node that is currently accessing it. Each memory module acts as a huge cache memory in which each block has a tag with the address and the state.

In this article, we explain the functionality, architecture, performance, and complexity of COMA systems. We also outline different COMA designs, compare COMA to traditional cache-coherent nonuniform memory access (NUMA) systems, and describe proposed improvements in NUMA systems that target the same performance obstacles as COMA.

COMA ORGANIZATION

In traditional NUMA multiprocessors, each node contains one or more processors with private caches and a memory module that is part of the global shared memory (see the sidebar, “Cache-Coherence and Latency-Tolerating Techniques”). A page allocated in the memory module of one node can be accessed by the processors of all other nodes. The physical address of the page specifies the node where the page is allocated. This is referred to as the page’s home node. The address of a memory block specifies the memory module and the location in that memory module where the block exists.

In these machines, fetching data from a remote memory module takes two to 10 times longer than fetching data from the local memory. Consequently, for an application to deliver high performance, the local memory module must satisfy a large fraction of its cache misses. This requires good placement of the program pages across the different nodes. If the program’s memory access patterns are too complicated for the operating system, compiler, or programmer to understand, placing the data in the memory module of the node that is most likely to access it is more difficult. In addition, when a page contains objects that are read and written by different processors, false sharing occurs, and this also complicates good page placement.

In COMA, the hardware can transparently eliminate a certain class of remote memory accesses. COMA does this by turning memory modules into large dynamic RAM (DRAM) caches called attraction memory (AM).[1] When a processor requests a block from a remote memory, the block is inserted in both the processor’s cache and the node’s AM. A block can be evicted from an AM if another block needs the space. Ideally, with this support, the processor dynamically attracts its working set into its local memory module. The data the processor is not accessing overflows and is sent to other memories. Because a large AM is more capable of containing the node’s current working set than a cache is, more of the cache misses are satisfied locally within the node.

The issues that need to be addressed in COMA include

- block localization,
- block replacement, and
- memory overhead.

0018-9162/99/$10.00 © 1999 IEEE

Block localization

When the memory modules act as caches, a block's address is a global identifier, not an indicator of its physical memory location. Just like a normal cache, the AM keeps a tag with the address and state of the currently stored memory block in each location. On a cache miss, the memory controller has to look up the local AM tags to determine whether or not the access can be serviced locally. If the block is not in the local AM, a remote request must be issued to find the block.

We need a mechanism to localize a block in the system so the processor can find a valid copy of the block when a miss occurs in the AM. The first COMA designs, the KSR-1 from Kendall Square Research and Data Diffusion Machine (DDM) from the Swedish Institute of Computer Science, were organized in a tree hierarchy of rings and buses, respectively.[1] These systems are called Hierarchical-COMA. The processors were connected to the leaves of the tree. Each level in the hierarchy included a directory with information about the status of the blocks extending from the leaves up to that level of the hierarchy. To find a block, the processing node issued a request that went to successively higher levels of the tree, starting at one leaf and potentially going all the way to the root. The process stopped at the level where the subtree contained the block.

Cache-Coherence and Latency-Tolerating Techniques

In a multiprocessor that uses a private cache on each processor, copies of a given shared-memory block can reside in multiple caches at the same time. A write to this block by one processor, therefore, requires a mechanism to prevent other processors from reading the old value from their cached copies; such a mechanism is called a *cache-coherence protocol*.[1]

The most commonly used cache-coherence technique is for the hardware to transparently invalidate all other cached copies of the same block. Since the copy in the writing processor's cache is now the only valid copy in the system, that processor can continue to write to the block locally without causing coherence problems. However, access to the block by any of the invalidated processors will result in a coherence or sharing miss. Application programs that write and read to shared variables a lot may suffer from numerous coherence misses.

In a scalable shared-memory multiprocessor, fetching data from the memory of a remote node involves significant latency. However, a number of latency-tolerating techniques can decrease the time the processor must stall in this situation. Most of these techniques are applicable to both NUMA and COMA systems. Three such techniques are prefetching, relaxed memory consistency, and multithreading.

- *Prefetching.* In prefetching, the processor fetches the data in a nonblocking manner, usually into the cache, before the processor needs it. Later, when the processor needs the data, it should find it in its cache. In software-controlled prefetching, the compiler or programmer inserts additional instructions in the code to perform the prefetching.[2] Hardware- controlled prefetching uses a mechanism that detects memory reference patterns and uses the patterns to automatically prefetch data.[3]
- *Relaxed memory consistency.* The memory consistency model specifies how the machine appears to handle memory operations. In the sequential-consistency model, memory operations appear to be carried out in the order specified by the program, and all processors observe the updates at the same time. The strict ordering imposed by this model makes it difficult for a processor to have multiple outstanding memory accesses at the same time. Unfortunately, if we disable the ability to overlap several memory accesses, the result is poor performance. We can, however, relax the memory consistency model and tolerate more overlapping of memory accesses except at synchronization points.[4] This relaxed memory ordering ultimately leads to higher performance because the latency of memory accesses can be hidden more. Examples of relaxed memory consistency models are release consistency, weak consistency, and relaxed memory ordering.
- *Multithreading.* Several forms of multithreading are possible. In one example, a processor tolerates the latency of a thread accessing memory by switching to another thread. In the multithreaded Tera architecture, the processor can execute as many as 128 threads concurrently.[5] Every cycle, execution switches from one thread to the next without penalty. While memory latency fully penalizes the performance of a single thread, the machine can accomplish high throughput.

References

1. P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer*, June 1990, pp. 12-24.
2. T. Mowry, "*Tolerating Latency through Software-Controlled Data Prefetching*," doctoral dissertation, Computer Systems Lab., Stanford Univ., Stanford, Calif., 1994.
3. F. Dahlgren and P. Stenstrom, "Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, Apr. 1996, pp. 385-398.
4. S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, Dec. 1996, pp. 66-76.
5. R. Alverson et al., "The Tera Compute System," *Proc. 1990 Int'l Conf. Supercomputing*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 1-6.

In more recent designs like Flat-COMA,[2] a fixed-location directory makes it easier to locate a block. In this design, each memory block has a directory entry in its home node. Memory blocks can freely migrate, but directory entries do not. Consequently, to locate a memory block, a processor interrogates the directory in the block's home node. The directory always knows the state and location of the block and can forward the request to the right node. Figure 1 illustrates the node organization in NUMA, Hierarchical-COMA, and Flat-COMA.

Block replacement

The AM acts as a cache, and blocks can be replaced from it. When a block is replaced in a plain cache, it is either overwritten (if it is unmodified) or written back to its home memory module, which guarantees a placeholder for the block.

Because blocks migrate in COMA, a block doesn't have a fixed backup location where it can be written if replacement from an AM occurs. Even an unmodified AM block can be the only copy of that memory block in the system, and it must not be overwritten on an AM replacement. Therefore, the system must keep track of the last copy of a block. As a result, when a modified or otherwise unique block is displaced from an AM, it must be relocated into another AM.

To guarantee that at least one copy of an unmodified block remains in the system, we can denote one of its multiple copies as the master copy. All other shared copies can be overwritten if replaced, but the master copy must always be relocated to another AM.

When a master copy or a modified block is relocated, the problem is deciding which node should take the block in its AM. If other nodes already have one or more other shared copies of the block, one of them becomes the master copy. Otherwise, another node must accept the block. This process is called block injection. In the DDM, the replacing processor sends requests to other nodes asking if they have space to host the block. Alternatively, we can simply force one node to accept the block. This, however, may lead to another block replacement. A solution proposed by Joe and Hennessy[3] is to relocate the new block to the node that supplied the block that caused the displacement in the first place.

Overall, these algorithms that handle the last copy of a block tend to complicate the cache-coherence protocol.

Memory overhead

A NUMA machine typically allocates all memory to application or operating system pages. COMA, however, leaves a portion of the memory unallocated to facilitate automatic data replication and migration. This unallocated space supports the replication of shared blocks across AMs. It also enhances block migration to the AMs of the referencing nodes because less block relocation traffic is required. Without unallocated space, every time we insert a block in the AM, another block would have to be relocated. The ratio between the application size and the total size of the AMs is called the memory pressure. If the memory pressure is 60 percent, 40 percent of the AM space is available for data replication. Both the relocation traffic and the number of AM misses increase with the memory pressure.[3] For a given memory size, choosing an appropriate memory pressure is a trade-off between the effect on page faults, AM misses, and relocation traffic.

ALTERNATIVE COMA DESIGNS

In the early Hierarchical-COMA designs, substantial latency occurred as the memory requests went up the hierarchy and then down to find the desired block. This latency can offset the potential gains of COMA relative to plain NUMA.[2]

Flat-COMA

Because Flat-COMA does not rely on a hierarchy to find a block, it can use any high-speed network. Its directory is distributed according to the physical address as in NUMA. The memory blocks can migrate, but the directory entries are fixed in their

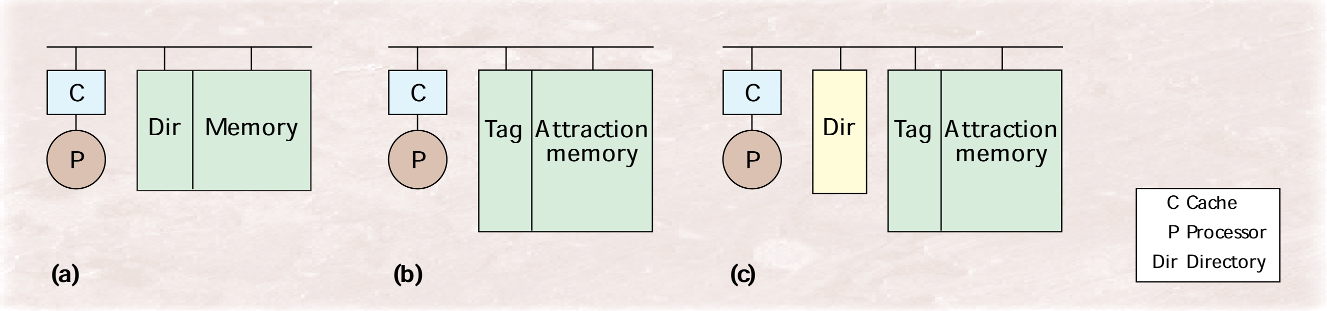

Figure 1. Node organization in (a) NUMA, (b) Hierarchical-COMA, and (c) Flat-COMA.

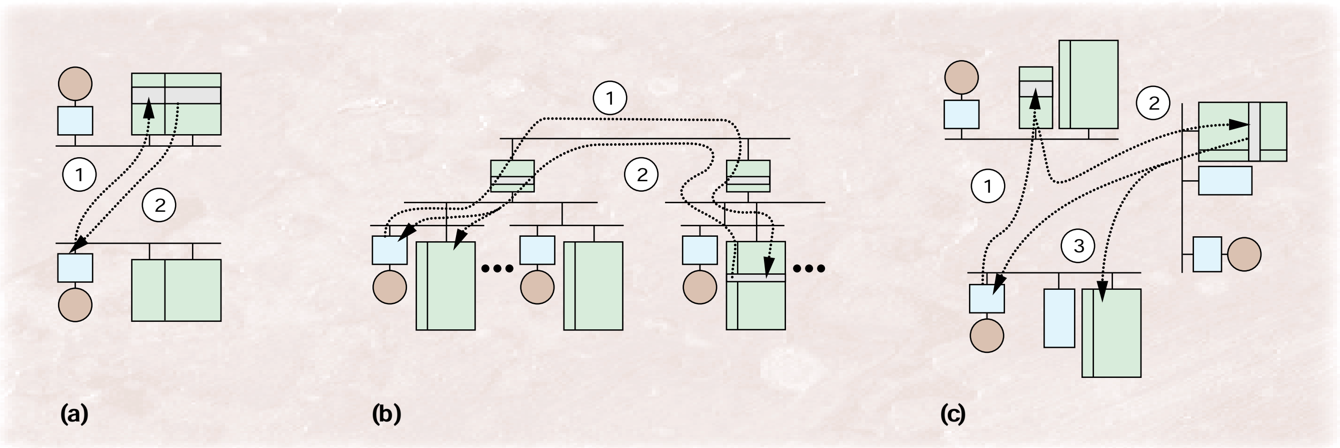

Figure 2. The paths followed by a remote memory request to an unwritten block in (a) NUMA, (b) Hierarchical-COMA, and (c) Flat-COMA. In Flat-COMA, the block in question has been displaced from its home AM.

home nodes. At a miss on a block in an AM, a request goes to the node that is keeping the directory information about the block. The directory redirects the request to another node if the home does not have a copy of the block. In Flat-COMA, unlike in NUMA, the home node may not have a copy of the block even though no processor has written to the block. The block has simply been displaced from the AM. Figure 2 illustrates the difference between a remote memory access in NUMA, Hierarchical-COMA, and Flat-COMA.

Simple-COMA

Hierarchical and Flat-COMA implement the AM block replacement and relocation mechanisms in hardware, which introduces complexity into the system. A new variety of COMA, called Simple-COMA (S-COMA)[4] transfers some of this complexity to software. The general coherence actions, however, are still maintained in hardware for performance reasons.

In S-COMA, the operating system sets aside space in the AM for incoming memory blocks on a page-granularity basis. The local memory management unit (MMU) has mappings only for pages in the local node, not for remote pages. On a node's first access a shared page that is already in a remote node, the processor suffers a page fault. The operating system then allocates a page frame locally for the requested data. Thereafter, the hardware continues with the request, including locating a valid copy of the block and inserting it, in the correct state, in the newly allocated page in the AM. The rest of the page remains unused until future requests to other lines of the page start filling it. Subsequent accesses to the block get their mapping directly from the MMU. There are no AM address tags to check to see if we are accessing the correct block.

Since the physical address used to identify a block in the AM is set up independently by the MMU in each node, two copies of the same block in different nodes are likely to have different physical addresses. Shared data needs a global identity so that different nodes can communicate. To this end, each node has a translation table that converts local addresses to global identifiers and vice versa.

Multiplexed Simple-COMA

One problem with S-COMA is memory fragmentation: S-COMA sets aside memory space in page-sized chunks, even if only one block of each page is present. This can cause programs to have inflated working sets that overflow the AM, inducing frequent page replacements and resulting in high operating-system overhead and poor performance.

Multiplexed Simple COMA (MS-COMA), a variation of S-COMA, seeks to eliminate this problem.[5] MS-COMA allows multiple virtual pages in a given node to map to the same physical page at the same time. This mapping is possible because all the blocks on a virtual page are not used at the same time. A given physical page can now contain blocks belonging to different virtual pages if each block has a virtual page ID. If two blocks belonging to different pages have the same page offset, they displace each other from the AM. The overall result is compression of the application's working set.

PERFORMANCE ISSUES

Comparing COMA machines to other scalable shared-memory systems provides insight into their performance.

Flat-COMA

The memory overhead of COMA makes a comparison between Flat-COMA and a traditional NUMA system inequitable. Therefore, to understand the performance of Flat-COMA, we compare it to NUMA-RC, a NUMA system with a large DRAM L3

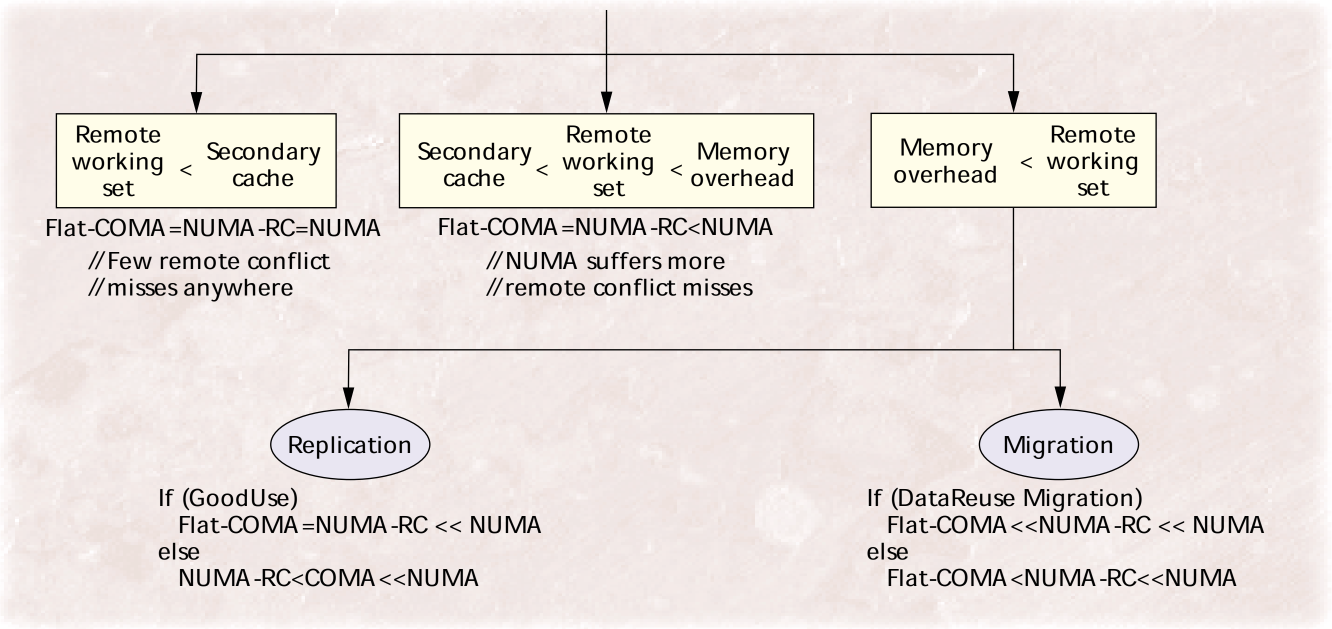
Remote working set < Secondary cache
Flat-COMA=NUMA-RC=NUMA
//Few remote conflict
//misses anywhere
Secondary cache < Remote working set < Memory overhead
Flat-COMA=NUMA-RC<NUMA
//NUMA suffers more
//remote conflict misses
Memory overhead < Remote working set
Replication
If (GoodUse)
Flat-COMA=NUMA-RC << NUMA
else
NUMA-RC<COMA<<NUMA
Migration
If (DataReuse Migration)
Flat-COMA<<NUMA-RC << NUMA
else
Flat-COMA<NUMA-RC<<NUMA

Figure 3. The number of remote conflict misses in NUMA, NUMA-RC, and Flat-COMA depends on the relative size of the remote working set, secondary cache, and memory overhead. In this schematic view, the less than, greater than, and equal to signs refer to the number of remote conflict misses.

(level-three) cache added to each node. This DRAM is called a remote cache (RC) because it stores remote data.[6] Other authors call this a cluster or network cache. A large RC increases the fraction of locally satisfied accesses because it intercepts requests that would otherwise access a remote node.

We use a model in which all machines have the same primary and secondary caches.[6] In addition, NUMA-RC and Flat-COMA have the same amount of memory overhead per node. In NUMA-RC, the memory overhead per node is the remote cache; in Flat-COMA, it is the extra memory added for data replication. Flat-COMA needs block tags for the entire AM, while NUMA-RC only needs block tags for the remote cache. However, the difference in overall size is negligible. Finally, we assume that an access to the remote cache in NUMA-RC and to the AM in Flat-COMA takes the same amount of time.

In all of these systems, accesses to remote memory modules caused by misses in the local memory hierarchy fall into three categories:

- *cold misses* are accesses to blocks never previously accessed by the node;
- *coherence misses* are accesses to blocks invalidated by the coherence mechanism; and
- *conflict misses* are accesses to blocks displaced from the local memory hierarchy because of overflow.

Our analysis focuses on conflict misses because cold and coherence misses for a given application are largely the same for NUMA, NUMA-RC, and Flat-COMA. The number of conflict misses, however, varies across architectures, depending on the number of misses serviced by the local memory hierarchy.

As Figure 3 shows, if the remote data in the working set of a processor fits in the secondary cache, a small number of remote conflict accesses occur in all three architectures. If, instead, the remote data overflows the secondary cache but fits in the memory overhead, the number of conflict accesses is likely to be high in NUMA and small in both NUMA-RC and Flat-COMA. When the remote working set overflows memory overhead, NUMA again has the highest number of conflict accesses. However, NUMA-RC can have more or fewer conflict accesses than Flat-COMA depending on the data access patterns.

Two major data access patterns, data replication and migration, can help us understand this case. Replication occurs when several processors access a data structure in a read-mostly manner. Migration occurs when one processor accesses remotely allocated data. Migration data includes both data accessed exclusively by one processor in the whole program and data accessed sequentially by one processor after another. Replication data structures replicate in the AMs of Flat-COMA and in the remote caches of NUMA-RC. A migration data structure needs to be in only one AM at a time.

The weight of the migration data in the working set determines whether Flat-COMA has fewer conflict accesses than NUMA-RC. Ideally, Flat-COMA only needs to fit the replication data in its memory overhead. Migration data does not use extra memory in Flat-COMA because it uses the same space in one AM that it frees up in another AM.

The memory overhead of NUMA-RC, however, houses both replication and migration data. If migration data dominates, NUMA-RC has more conflict accesses than Flat-COMA. When replication data dominates, if the memory overhead area of the AM is used well (denoted GoodUse in Figure 3), the two systems have about the same number of conflict misses. However, because it is harder for Flat-COMA to use the memory overhead area well,[6] Flat-COMA often suffers more conflict misses than NUMA-RC under replication data.

With migration data, the savings in misses in Flat-COMA depend on the type of migration behavior. The savings of Flat-COMA are relatively higher when a processor accesses a data structure many times before it relinquishes it to another processor. We call this DataReuse migration. If a data structure changes owner processors very often, the savings of Flat-COMA are reduced because it cannot eliminate the misses in the transient phase when the data changes owners.

In addition to the relative number of remote conflict accesses, the difference in performance between Flat-COMA and NUMA-RC is affected by the amount of processor stall time the average misses induce. This stall time tends to be larger in Flat-COMA than in NUMA-RC for several reasons. First, remote cold and remote conflict accesses cost more in Flat-COMA. While these accesses are always satisfied in two network hops in NUMA-RC, they may need three hops in Flat-COMA because the AM in the home node displaces blocks to free up space. Second, remote coherence accesses also cost more in Flat-COMA. One reason is that dirty data is displaced less frequently from the large AMs of Flat-COMA than from the smaller remote caches of NUMA-RC. As a result, the home is less likely to be up-to-date.

Overall, while it tends to suffer fewer remote conflict accesses than NUMA-RC, especially when migration data dominates, Flat-COMA's performance may not necessarily be higher because the average memory access is more expensive.

Simple COMA

In S-COMA, AM space is set aside in page-sized chunks, even though only one block is brought in on a miss. In contrast, Flat-COMA only sets aside space for the missed block. The space allocation in S-COMA may cause memory fragmentation and degrade overall performance due to the operating system page management overhead. MS-COMA addresses this problem by multiplexing several virtual pages into a single physical page at the same time, compressing the working set.

The model shown in Figure 4 uses two aspects of application access patterns. The first aspect is the number of memory block conflicts in the local memory of Flat-COMA or in the remote cache of NUMA-RC. The number of such conflicts is a function of the application's temporal and spatial locality and the mapping of virtual addresses to physical memory.

The second aspect is the degree of spatial locality of the accesses at the page level, which refers to the fraction of each page that is used. There are two possible scenarios here: In the first, spatial locality is low and the size of the page working set is larger than the local memory. The second scenario comprises all other situations. Each cell in Figure 4 indicates which architectures have high performance under given characteristics. Specifically, if there are few block conflicts (top row), Flat-COMA and NUMA-RC both work well. Otherwise, they have poor performance (bottom row). S-COMA is less susceptible to these block conflicts because it can allocate a page in several nodes with different physical addresses.

On the other hand, if spatial locality at the page level is low and the page working set overflows the local memory (left column), S-COMA has poor performance. Even though it doesn't use much of the memory, S-COMA consumes CPU time because the operating system has to repeatedly allocate and deallocate pages. Under other conditions (right column), S-COMA performs well. These conditions do not affect Flat-COMA and NUMA-RC because their memory allocation unit is a block. They don't waste memory when spatial locality at the page level is poor.

MS-COMA is designed for use when neither Flat-COMA nor S-COMA works well. As in S-COMA, each node in MS-COMA is free to allocate a page in

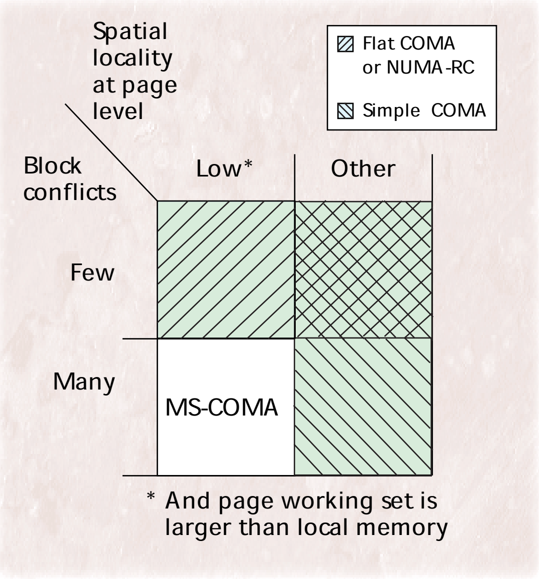
Spatial locality at page level
Block conflicts
Flat COMA or NUMA-RC
Simple COMA
Low*
Other
Few
Many
MS-COMA
* And page working set is larger than local memory

Figure 4. Simplified plot compares the performance of hardware-based systems (NUMA-RC and Flat-COMA) to software-based systems (S-COMA and MS-COMA).

any local location, thereby reducing block conflicts. In addition, MS-COMA uses page multiplexing to compress the page working set when it would overflow the local memory because of poor spatial locality at the page level.

COMA has an advantage over NUMA-RC in some situations and vice versa. Consequently, supporting COMA and NUMA on a per-page basis is an appealing approach.

ALTERNATIVES TO COMA

Two alternatives to COMA are using the operating system to migrate pages in NUMA-RC and combining NUMA-RC and COMA in a single machine.

Page migration in NUMA-RC

To reduce the number of remote memory accesses in NUMA-RC, the operating system can migrate pages to the nodes where they are most actively used.[7] As we have seen, Flat-COMA has a potential performance advantage over NUMA-RC when migration data dominates. The difference is that Flat-COMA can reuse the unused space left behind by the migration data, while NUMA-RC cannot. However, if the migration object is about the same size as a page, the operating system can explicitly trigger page migration in NUMA-RC. As a result, NUMA-RC can mimic the behavior of COMA.

A page migration mechanism must be able to identify pages that are candidates for migration and decide when and where to migrate them. When migration takes place, the mechanism must copy the page, update the page table, and invalidate the old translation from all the translation look-aside buffers (TLBs).

The SGI Origin multiprocessor includes hardware and software mechanisms to support page migration. Specifically, a per-page and per-node memory reference counter keeps track of both remote and local memory accesses to a page. When the counter reaches a threshold in terms of the difference between remote and local memory accesses, the system migrates the page. The system has a block copy engine to copy the page efficiently and a mechanism to reduce the cost of TLB changes.

However, each page migration creates significant overhead. Therefore, we must identify the systems and applications for which page migration will substantially boost performance. For NUMA-RC, we can expect a significant improvement for applications that have roughly page-size objects accessed in DataReuse migration, where good static page placement is difficult, and where the remote working set is significantly larger than the remote cache.[8]

Combining NUMA-RC and COMA

COMA has an advantage over NUMA-RC in some situations and vice versa. Consequently, supporting COMA and NUMA on a per-page basis is an appealing approach. To do that, we need mechanisms to detect which of the two schemes is best for each page or block so that we can change from one to the other as needed.

Reactive NUMA supports NUMA-RC and S-COMA on a per-page basis.[9] In this organization, all pages start in NUMA mode. The operating system uses information about the number of refetches issued by a node to blocks of a page to switch the page to S-COMA mode. A *refetch* is an access that brings a block that was displaced from the local memory hierarchy due to overflow, back into a node's local memory hierarchy. A hardware counter tracks the number of refetches from a node to a page. When the counter reaches a specified threshold, the operating system switches the page to S-COMA mode for that node. The WildFire multiprocessor from Sun Microsystems uses this principle.[10]

In Reactive-NUMA, all pages start in the upper row of Figure 4. When a page is moving toward the lower row, the operating system switches it to S-COMA mode. The operating system allocates a local page frame for the page in the node and provides a translation between the new local address and the page's global address.[9] Reactive-NUMA uses triggers other than a low degree of spatial locality at the page level to revert pages back to NUMA.

The IBM Prism architecture also supports NUMA-RC and S-COMA on a per-page basis.[11] In this organization, pages start in S-COMA, and they are reconfigured into NUMA based on runtime information.

Overall, at this time, there are many open problems in the area of COMA architectures. COMA's advantage is the transparent fine-grain replication and migration of data, adapting dynamically to the application's reference patterns. However, architectural and application trends may affect COMA's potential as a viable alternative in the future.

We expect the relative cost of a remote memory access to increase in the future. Consequently, reducing the fraction of remote memory accesses and supporting data replication and migration in a way that is transparent to the programmer will become more important.

Keeping the cache-coherence protocol simple is also an important consideration. This favors a design based on a traditional NUMA-RC or even S-COMA over Flat-COMA. Larger, more complicated remote caches might provide a way to capture a large remote working set without COMA support.[12] Hybrid machines that combine the performance of COMA with the relative simplicity of NUMA-RC are likely to become the preferred design.

The relative performance of COMA and NUMA-

RC depends heavily on the type of application being run. If the application is tuned for the memory hierarchy of a NUMA machine, NUMA-RC will likely perform well. However, if the application is not tuned and requires many remote memory accesses, COMA will likely perform better.

Unfortunately, most of the work addressing COMA-NUMA trade-offs uses scientific or engineering applications that, to some extent, are tuned for a NUMA memory hierarchy. Consequently, the currently available data could be biased in favor of the more traditional architectures that do not support automatic data migration and replication. New application domains like databases and multimedia need to be explored. ❖

Acknowledgments

This work was carried out while Fredrik Dahlgren was at Chalmers University of Technology. His research is supported by the Swedish National Board for Industrial and Technical Development (NUTEK) under contract P10502, by the Swedish Research Council on Engineering Science (TFR) under contract 97-593, and by Sun Microsystems. Josep Torrellas' research is supported by US National Science Foundation Young Investigator Award MIP-9457436, DARPA Contract DABT63-95-C-0097, and by IBM and Intel.

References

1. E. Hagersten, A. Landin, and S. Haridi, "DDM—A Cache-Only Memory Architecture," *Computer*, Sept. 1992, pp. 44-54.
2. P. Stenstrom, T. Joe, and A. Gupta, "Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures," *Proc. 19th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 80-91.
3. T. Joe and J. Hennessy, "Evaluating the Memory Overhead Required for COMA Architectures," *Proc. 21st Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 82-93.
4. A. Saulsbury et al., "An Argument for Simple COMA," Proc. *1st Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 276-285.
5. S. Basu and J. Torrellas, "Enhancing Memory Use in Simple COMA: Multiplexed Simple COMA," *Proc. 4th Int'l Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 152-161.
6. Z. Zhang and J. Torrellas, "Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA," *Proc. 3rd Int'l Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 272-281.
7. B. Verghese et al., "Operating System Support for Improving Data Locality on CC-NUMA Computer Servers," *Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1996, pp. 279-289.
8. V. Soundararajan et al., "Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors," *Proc. 25th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 342-355.
9. B. Falsafi and D. Wood, "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, pp. 229-239.
10. E. Hagersten and M. Koster, "WildFire: A Scalable Path for SMPs," *Proc. 5th Int'l Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 172-181.
11. K. Ekanadham et al., "PRISM: An Integrated Architecture for Scalable Shared Memory," *Proc. 4th Int'l Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 140-151.
12. A. Moga and M. Dubois, "The Effectiveness of SRAM Network Caches in Clustered DSMs," *Proc. 4th Int'l Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 103-112.

***Fredrik Dahlgren** is a research and design engineer at Ericsson Mobile Communications, Lund, Sweden. Dahlgren received an MSc and a PhD in computer engineering from Lund University. His research interests include computer systems architecture for future mobile terminals, with a focus on energy efficiency. He is a member of the IEEE Computer Society. Contact Dahlgren at fredrik.dahlgren@ecs.ericsson.se.*

***Josep Torrellas** is an associate professor in the computer science department at the University of Illinois at Urbana-Champaign. His research interests include new processor and memory technologies, software, and organizations to build uni- and multiprocessor architectures. Torrellas received a PhD in electrical engineering from Stanford University. He has received the NSF Research Initiation Award, the NSF Young Investigator Award, and several IBM Partnership Awards. He is a member of the IEEE and ACM and serves as vice-chair of the IEEE Technical Committee on Computer Architecture. Contact Torrellas at torrella@cs.uiuc.edu.*