Cache-Only Memory Architecture

Byline

Josep Torrellas
Computer Science Department, University of Illinois at Urbana-Champaign
torrella@illinois.edu

Synonyms

COMA.

Definition

A Cache-Only Memory Architecture (COMA) is a type of cache-coherent non-uniform memory access (CC-NUMA) architecture. Unlike in a conventional CC-NUMA architecture, in a COMA, every shared-memory module in the machine is a cache, where each memory line has a tag with the line's address and state. As a processor references a line, it transparently brings it to both its private cache(s) and its nearby portion of the NUMA shared memory (*Local Memory*) — possibly displacing a valid line from its local memory. Effectively, each shared-memory module acts as a huge cache memory, giving the name COMA to the architecture. Since the COMA hardware automatically replicates the data and migrates it to the memory module of the node that is currently accessing it, COMA increases the chances of data being available locally. This reduces the possibility of frequent long-latency memory accesses. Effectively, COMA dynamically adapts the shared data layout to the application's reference patterns.

Discussion

1 Basic Concepts

In a conventional CC-NUMA architecture, each node contains one or more processors with private caches and a memory module that is part of the NUMA shared memory. A page allocated in the memory module of one node can be accessed by the processors of all other nodes. The physical page number of the page specifies the node where the page is allocated. Such node is referred to as the *Home Node* of the page. The physical address of a memory line includes the physical page number and the offset within that page.

In large machines, fetching a line from a remote memory module can take several times longer than fetching it from the local memory module. Consequently, for an application to attain high performance, the local memory module must satisfy a large fraction of the cache misses. This requires a good placement of the program pages across the different nodes. If the program's memory access patterns are too complicated for the software to understand, individual data structures may not end up being placed in the memory module of the node that access them the most. In addition, when a page contains data structures that are read and written by different processors, it is hard to attain a good page placement.

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

There are three issues that need to be addressed in COMA, namely finding a line, replacing a line, and dealing with the memory overhead. In the rest of this article, we first describe these issues, then outline different COMA designs and finally point to further readings.

1.1 Finding a Memory Line

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

COMA machines have a mechanism to locate a line in the system, so that the processor can find a valid copy of the line when a miss occurs in the local AM. Different mechanisms are used by different classes of COMA machines.

One approach is to organize the machine hierarchically, with the processors at the leaves of the tree. Each level in the hierarchy includes a directory-like structure, with information about the status of the lines present in the subtree extending from the leaves up to that level of the hierarchy. To find a line, the processing node issues a request that goes to successively higher levels of the tree, potentially going all the way to the root. The process stops at the level where the subtree contains the line. This design is called Hierarchical COMA [2, 7].

Another approach involves assigning a home node to each memory line, based on the line's physical address. The line's home has the directory entry for the line. Memory lines can freely migrate, but directory entries do not. Consequently, to locate a memory line, a processor interrogates the directory in the line's home node. The directory always knows the state and location of the line and can forward the request to the right node. This design is called Flat COMA [12].

1.2 Replacing a Memory Line

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

A memory line in COMA does not have a fixed backup location where it can be written to if it gets displaced from an AM. Moreover, even an unmodified line can be the only copy of that memory line in the system, and it must not be lost on an AM displacement. Therefore, the system must keep track of the last copy of a line. As a result, when a modified or otherwise unique line is displaced from an AM, it must be relocated into another AM.

To guarantee that at least one copy of an unmodified line remains in the system, one of the line's copies is denoted as the *Master* copy. All other shared copies can be overwritten if displaced, but the master copy must always be relocated to another AM. When a master copy or a modified line is relocated, the problem is deciding which node should take the line in its AM. If other nodes already have one or more other shared copies of the line, one of them becomes the master copy. Otherwise, another node must accept the line. This process is called *Line Injection*.

Different line injection algorithms are possible. One approach is for the displacing node to send requests to other nodes asking if they have space to host the line [7]. Another approach is to force one node to accept the line. This, however, may lead to another line displacement. A proposed solution is to relocate the new line to the node that supplied the line that caused the displacement in the first place [8].

1.3 Dealing with Memory Overhead

A CC-NUMA machine can allocate all memory to application or 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 lines across AMs. It also enhances line migration to the AMs of the referencing nodes because less line relocation traffic is needed.

Without unallocated space, every time a line is inserted in the AM, another line would have to be relocated.

The ratio between the allocated data size and the total size of the AMs is called the *Memory Pressure*. If the memory pressure is 80 percent, then 20 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 [8]. 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.

2 Different Cache-Only Memory Architecture Designs

2.1 Hierarchical COMA

The first designs of COMA machines follow what has been called Hierarchical COMA. These designs organize the machine hierarchically, connecting the processors to the leaves of the tree. These machines include the KSR-1 [2] from Kendall Square Research, which has a hierarchy of rings, and the Data Diffusion Machine (DDM) [7] from the Swedish Institute of Computer Science, which has a hierarchy of buses.

Each level in the tree hierarchy includes a directory-like structure, with information about the status of the lines extending from the leaves up to that level of the hierarchy. To find a line, the processing node issues a request that goes to successively higher levels of the tree, potentially going all the way to the root. The process stops at the level where the subtree contains the line.

In these designs, substantial latency occurs as the memory requests go up the hierarchy and then down to find the desired line. It has been argued that such latency can offset the potential gains of COMA relative to conventional CC-NUMA architectures [12].

2.2 Flat COMA

A design called Flat COMA makes it easy to locate a memory line, by assigning a home node to each memory line [12] — based on the line's physical address. The line's home has the directory entry for the line, like in a conventional CC-NUMA architecture. The memory lines can freely migrate, but the directory entries of the memory lines are fixed in their home nodes. At a miss on a line in an AM, a request goes to the node that is keeping the directory information about the line. The directory redirects the request to another node if the home does not have a copy of the line. In Flat COMA, unlike in a conventional CC-NUMA architecture, the home node may not have a copy of the line even though no processor has written to the line. The line has simply been displaced from the AM in the home node.

Because Flat COMA does not rely on a hierarchy to find a block, it can use any high-speed network.

2.3 Simple COMA

A design called Simple COMA (S-COMA) [10] transfers some of the complexity in the AM line displacement and relocation mechanisms to software. The general coherence actions, however, are still maintained in hardware for performance reasons. Specifically, 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. When a node accesses for the first time 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 line. Thereafter, the hardware continues with the request, including locating a valid copy of the line and inserting it, in the correct state, in the newly allocated page in the local AM. The rest of the page remains unused until future requests to other lines of the page start filling it. Subsequent accesses to the line get their mapping directly from the MMU. There are no AM address tags to check to see if we are accessing the correct line.

Since the physical address used to identify a line in the AM is set up independently by the MMU in each node, two copies of the same line 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.

2.4 Multiplexed Simple COMA

S-COMA sets aside memory space in page-sized chunks, even if only one line of each page is present. Consequently, S-COMA suffers from memory fragmentation. 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) [1] eliminates this problem by allowing 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 lines on a virtual page are not used at the same time. A given physical page can now contain lines belonging to different virtual pages if each line has a short virtual page ID. If two lines belonging to different pages have the same page offset, they displace each other from the AM. The overall result is a compression of the application's working set.

3 Further Readings

There are several papers that discuss COMA and related topics. Dahlgren and Torrellas present a more in-depth survey of COMA machine issues [3]. There are several designs that combine COMA and conventional CC-NUMA architecture features, such as NUMA with Remote Caches (NUMA-RC) [9], Reactive NUMA [5], Excel-NUMA [14], the Sun Microsystems' WildFire multiprocessor design [6], the IBM Prism architecture [4], and the Illinois I-ACOMA architecture [13]. A model for comparing the performance of COMA and conventional CC-NUMA architectures is presented by Zhang and Torrellas [15]. Soundarajan *et al* [11] describe the tradeoffs related to data migration and replication in CC-NUMA machines.

Related Entries

Cache-Coherent Non-Uniform Memory Access (CC-NUMA) architecture.

References

[1] S. Basu and J. Torrellas. Enhancing Memory Use in Simple Coma: Multiplexed Simple Coma. In *International Symposium on High-Performance Computer Architecture*, February 1998.

[2] H. Burkhardt et al. Overview of the KSR1 Computer System. Technical Report 9202001, Kendall Square Research, Waltham, MA, February 1992.

[3] F. Dahlgren and J. Torrellas. Cache-Only Memory Architectures. *IEEE Computer Magazine*, June 1999.

[4] K. Ekanadham, B.-H. Lim, P. Pattnaik, and M. Snir. PRISM: An Integrated Architecture for Scalable Shared Memory. In *International Symposium on High-Performance Computer Architecture*, February 1998.

[5] B. Falsafi and D. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *International Symposium on Computer Architecture*, June 1997.

[6] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *International Symposium on High-Performance Computer Architecture*, January 1999.

[7] E. Hagersten, A. Landin, and S. Haridi. DDM – a Cache-Only Memory Architecture. *IEEE Computer*, pages 44–54, September 1992.

[8] T. Joe and J. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *International Symposium on Computer Architecture*, pages 82–93, April 1994.

[9] A. Moga and M. Dubois. The effectiveness of SRAM network caches in clustered DSMs. In *International Symposium on High-Performance Computer Architecture*, February 1998.

[10] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *International Symposium on High-Performance Computer Architecture*, pages 276–285, January 1995.

[11] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In *International Symposium on Computer Architecture*, June 1998.

[12] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *International Symposium on Computer Architecture*, pages 80–91, May 1992.

[13] J. Torrellas and D. Padua. The Illinois Aggressive Coma Multiprocessor Project (I-ACOMA). In *Symposium on the Frontiers of Massively Parallel Computing*, October 1996.

[14] Z. Zhang, M. Cintra, and J. Torrellas. Excel-NUMA: Toward Programmability, Simplicity, and High Performance. *IEEE Transactions on Computers, Special Issue on Cache Memory*, February 1999.

[15] Z. Zhang and J. Torrellas. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. In *International Symposium on High-Performance Computer Architecture*, pages 272–281, February 1997.