Upcoming Architectural Advances in DSM Machines and Their Impact on Programmability*

Josep Torrellas
Computer Science Department
University of Illinois at Urbana-Champaign, USA
http://iacoma.cs.uiuc.edu

Abstract

Major advances in the architecture of Distributed Shared-Memory (DSM) multiprocessors are in the offing, largely driven by the increasing integration of transistors on a chip. These advances promise large performance improvements. However, to truly exploit them is likely to require an increase in application tuning efforts.

1 Introduction

Distributed Shared-Memory (DSM) multiprocessors are taking an increasing share of the market for medium- and large-scale high-performance parallel processing. These machines, built out of fast off-the-shelf microprocessors, and sometimes even networks and nodes, exploit mass-market architectural designs. This characteristic makes them more cost-effective platforms than traditional vector-based supercomputers for many parallel applications.

In addition, it is argued that the simplicity of the shared-memory abstraction delivers a programmable environment. It is well-known, of course, that tuning applications for performance in these machines is often a time-consuming effort, largely due to the complexity of the memory hierarchy. Overall, however, the combination of cost-effective hardware and relative ease of programming can well make DSM multiprocessors the machines of choice for high-end parallel computing in the near future.

The architecture of these machines, however, is continuously impacted by rapid advances in architectural technologies. These advances are likely to result in large performance improvements in the next few years. Unfortunately, it is less clear that the considerable cost of tuning applications for performance will be reduced.

The major drive behind these new architectural technologies is the increasing integration of transistors on a chip, as documented by the Semiconductor Industry Association [16]. This trend will not ensure ever-faster processor frequencies, as technological constraints are believed to bound the frequencies of traditional-technology processors to a few GHz. Instead, the higher performance will come from the fact that high transistor density will enable both more system integration and more sophisticated designs. In addition, there are other evolutionary approaches that can also improve the performance of DSM designs.

*This work was supported in part by NSF under grants Young Investigator Award MIP-9457436, ASC-9612099 and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613 and gifts from Intel and IBM.

In this paper, we examine all these architectural technologies. We start by examining some of the evolutionary sources of architectural improvement (Section 2). Then, we consider the new architectural technologies (Section 3): the trend towards more integration (Section 3.1) and more sophistication (Section 3.2). Finally, we discuss how all this affects programmability (Section 4).

2 Evolutionary Architectural Improvements

Many subsystems of current DSM architectures can be gainfully improved with evolutionary changes, especially those that are related to or deal with the long-latency remote memory accesses. Two examples of such subsystems are the remote cache and the support for data migration and replication.

The remote cache, also called cluster or network cache, is a multi-megabyte DRAM tertiary cache that is added to each node. Its purpose is to hold the fraction of the data in the working set whose home is remote [25]. Remote caches intercept processor requests and, therefore, reduce the number of long-latency remote accesses. This fact makes them effective in speeding-up some applications. However, they can also slow down the remaining remote accesses.

The organization of the remote cache can be optimized so that the remote cache behaves like a much larger cache without the associated space and timing costs. Possible approaches are to combine the remote cache with a victim cache [7, 10], to combine it with a page cache [3], or to expand the remote cache over the local memory as in Excel-NUMA [24].

The victim cache is a small, often fully-associative cache designed to hold frequently-accessed lines that, due to conflicts in the remote cache, would otherwise be continuously displaced. The page cache is a cache that reserves space for several important pages. As a result, these pages do not suffer any cache conflicts. Their lines are always locally available unless they have been invalidated. Finally, the Excel-NUMA design allows data displaced from the remote cache to be temporarily stored in unused space in the local memory. If referenced, such data can then be retrieved locally, in a manner as if the local remote cache had been logically expanded over the local main memory.

It is possible that even a remote cache enhanced with all these supports may not be able to hold the whole remote working set. To address this problem, there are other techniques for data migration and replication across memories. With these techniques, the whole memory in the node becomes like a cache, able to hold remote data. Page migration and replication can be supported in software by the operating system [21]. Alternatively, memory line migration and replication can be supported in hardware, in which case the architecture is known as a Cache-Only Memory Architecture (COMA).

While different COMA flavors exist, the most implementable type is called Simple COMA [15]. Both IBM [2] and Sun Microsystems [5] have announced systems with Simple-COMA support. This approach is attractive because it minimizes the number of remote memory accesses due to conflicts in a transparent and relatively inexpensive manner.

Finally, there are indications that better results may be obtained by a selective combination of support for COMA and traditional Cache Coherent Non-Uniform Memory Access (CC-NUMA) architecture on a page-by-page basis [2, 3]. A given page may be initially set up under the COMA or under the CC-NUMA protocol. Then, as execution proceeds, hardware that monitors the patterns of access to the page decides whether the page should change its protocol. A page may change its protocol several times during execution. The complete process is expected to work transparently to the user.

Overall, all these techniques are based on evolutionary architectural changes, and try to improve the performance of DSM machines by minimizing the impact of long-latency remote memory accesses.

3 New Architectural Technologies

3.1 More System Integration

One direct consequence of the higher numbers of transistors that can be placed on a chip is that future DSM multiprocessors are likely to be built out of highly-integrated chips or packages.

3.1.1 Integration of Processor and Memory. One major trend is to increasingly integrate processor and main memory. Initially, this trend will imply building packages that can include a few separate processor and DRAM memory chips, in what is called Multi-Chip Module (MCM) technology [18]. Eventually, processor and DRAM memory are likely to share the same chip, in what is called Merged Logic and DRAM (MLD) technology [6]. In any case, the trend is toward DSM multiprocessor nodes with a lot of storage that is very close to the processor. For example, in a few years, we may have many hundreds of Mbytes of DRAM within a round trip access of 20-40 ns from the processor. On the other hand, the ratio of remote to local memory access latency will probably increase a few times relative to current values.

MCM technology is likely to remain more feasible than MLD technology in the next few years. In MCM, processor and memory chips are fabricated with different processes, which are optimized for speed and density respectively. Relatively little technological change is necessary, while inter-chip communication, which does not need to leave the package, is faster than in traditional systems. In MLD technology, on the other hand, a single fabrication process should be able to combine high-density DRAM memory with high-speed processor logic. Currently, the result is modest values in either memory density or processor speed. For example, it is now possible to include in a chip the memory that comes with a low-end PC (32-64 Mbytes) and a simple processor cycling at around 200 MHz. However, in a not far-off future, a chip will likely fit the memory that now comes with a high-end server and faster processors.

Architectures that integrate processor and memory in the same chip are called processor-in-memory (PIM) or intelligent memory [8, 12, 14] architectures. An important challenge for these architectures is how to ensure that the processor exploits the large bandwidth that the on-chip memory can potentially provide. Even a wide-issue superscalar can only sustain so many memory accesses per cycle. There is the danger that a PIM chip performs little better than a processor with a large on-chip secondary cache [1]. Consequently, it has been argued that, for applications to fully exploit PIM architectures, the processor design and even the computational model need to be revised significantly [13]. For example, the chip may have to include several interconnected processors and maybe some special hardware to perform memory-intensive computations.

3.1.2 Integration of Multiple Compute Engines. Alternatively, instead of integrating processor and memory, a second trend is to integrate multiple compute engines in a chip. In a few years, one such chip may include, for example, 4 low-issue superscalar processors with their private primary caches, an interconnect and a shared secondary cache. Many, highly varied designs have been proposed, which range from a simple chip multiprocessor with several low-issue processors [11], to a chip with several tightly-coupled compute engines with sophisticated hardware for register communication and speculative parallelization [17] (Section 3.2), to a Simultaneous Multithreaded (SMT) processor [20]. Roughly speaking, these architectures can be classified based on the degree of resource sharing between execution threads and the degree of coupling between compute engines.

The degree of resource sharing between threads varies across architectures. Many resources can be shared, including the instruction fetch window, the functional units and the primary cache. In some architectures, like SMT, threads share all the resources mentioned. In other architectures, like the simple chip multiprocessor mentioned before, they share none of those. Resource sharing increases the fraction of the chip that is usable in the serial section, when only one thread is running. For example, in a SMT, most of the chip is usable in the serial section. However, in a simple chip multiprocessor, except for the secondary cache, only about $1/N$ of the chip is usable, where N is the number of processors. Unfortunately, resource sharing introduces centralization and contention. High-frequency chip designs try to avoid centralization because driving signals to the centralized point involves multi-cycle latencies.

A second axis of classification is the degree of coupling between compute engines. A simple chip multiprocessor, where the different processors can only communicate through the secondary cache, has a low degree of coupling. However, many hardware additions can increase the coupling between compute engines and make the chip function more as an ensemble. For example, the chip may have special hardware for direct register communication between processors [17], support for speculative parallelization [4, 9, 19], or support for fast synchronization between compute engines. While all these features can speed-up applications running on a single chip, they are much harder to utilize effectively when the chip is used as a node of a DSM multiprocessor.

Overall, the trend toward the integration of multiple compute engines in a chip will enable clusters of processors within a DSM machine. Within these clusters, it is possible to share data in a fine-grain manner. In addition, if the compute engines within the chip are tightly coupled, intra-cluster computation and communication can be much faster. Whether this clustered DSM multiprocessor organization can be exploited without unduly complicating application design is an open question.

3.2 More Sophistication

A second way of using the increasing numbers of on-chip transistors is not to integrate bigger systems but to add additional support to perform certain functions better. While many improvements are possible, a especially important issue today is to use extra hardware to predict future events and, based on the prediction, speculatively execute future activity now. By executing future activity without waiting for the outcome of the predicted event, pipelining can be exploited better.

Examples of future events that are often predicted are branch outcomes and the non-existence of dependences between memory accesses. In the first example, the corresponding speculative execution is to execute instructions past the branch before knowing the outcome of the branch. In the second example, the speculative execution may be to perform a load before a previous store to possibly the same location is executed.

The support required for prediction is a combination of state that keeps past history information and extra computation required to come up with the prediction. The main support required for speculatively execution is the ability to discard or undo the work if the prediction is proven incorrect. In general, all these actions are invisible to the programmer.

One type of speculation that is especially relevant to DSM machines is speculative parallelization. This technique involves the parallel execution of code that the compiler could not identify as either parallel, or clearly not parallel. While speculative parallelization can be done at the chip-multiprocessor level [4, 9, 19], our focus is DSM multiprocessors [22, 23]. In both cases, the idea is to extend the hardware support for cache coherence to detect, at run-time, a dependence violation. If no violation is detected, parallel execution finishes successfully. However, as soon as a dependence violation is detected, execution rolls back to either the very beginning of the speculative code or to some intermediate state. Then, parallel or serial execution is resumed. This whole process is transparent to the programmer.

The significance of speculative parallelization in DSM multiprocessors is that important, Large-scale fully-parallel loops and codes often have dependence structures that cannot be analyzed by the compiler. Therefore, these codes have to be run serially. With the speculative hardware, they can run in parallel. Similar statements can be made for codes with only a few data dependences. An example of such codes are loops that access arrays with subscripted subscripts, where the subscript function or array is read-in as an input to the program. These codes are frequent in the domains of sparse matrix computations, domain decomposition, molecular dynamics, molecular biology and image processing.

Overall, with so many available transistors, some form of speculative parallelization may well be developed. The resulting DSM multiprocessors will speed-up applications without the need to re-write the code to make the data access patterns visible to the compiler.

4 Implications on Programmability

Performance-tuning applications in DSM machines often focuses on making effective use of the memory hierarchy. Since most of the architectural trends described in this paper are related to the memory hierarchy, it is natural that they impact the programmability of DSM machines. In this section, we explore how they do it.

We consider first the remote caches, COMA support, operating system-driven page migration and replication and the other evolutionary improvements of Section 2. These schemes try to reduce the number of long-latency remote memory accesses by dynamically changing the data that resides in the limited-sized local memory hierarchy reservoir. Consequently, because they minimize the effect of the remote memory in a mode that is largely transparent to the programmer, they help programmability. Among these schemes, those that use the page as the basic protocol unit, like operating system-driven page migration and replication or selective COMA/CC-NUMA adaptation, may need programmer intervention for best performance. Indeed, the programmer can tell the compiler what data structures have similar behavior and, therefore, should be packed in the same page. If programmer intervention is required for acceptable performance, programmability is hurt.

MCM- and PIM-based processor-memory integration as described in Section 3.1.1 will increase to many hundreds of Mbytes the amount of DRAM memory that can be accessed by the processor in a round trip of very few tens of nanoseconds. As a result, processors will be able to keep vast working sets within very close reach. However, it is unclear how processors will effectively exploit all the bandwidth that so much close-by memory can provide. If, as indicated in Section 3.1.1, the bandwidth is exploited by placing in each node several interconnected processors and maybe some special hardware for memory-intensive computations, programs will have to be re-written and, therefore, programmability will certainly be hurt. In addition, the fact that the ratio of remote to local memory access latency is likely to increase relative to current values does not help either.

As indicated in Section 3.1.2, the use of chip multiprocessors in DSM machines enables a clustered DSM multiprocessor organization. Unfortunately, to exploit this organization, the application needs to be re-written, possibly to include two different algorithms: an intra- and an inter-cluster one. This is especially true if the compute engines within each chip are tightly coupled (Section 3.1.2). Clearly, such an approach hurts programmability.

Finally, speculative parallelization as discussed in Section 3.2 is one technique that will help programmability. If the machine is equipped with such support, it can safely run in parallel loops and codes that the compiler cannot analyze, without requiring the programmer to rewrite them to make the data access patterns visible to the compiler. Speculative parallelization is transparent to the programmer.

Overall, therefore, we can see that most of the architectural technologies described are not going to make tuning the application for performance any easier. In general, if we do not want to exploit the new features, the code does not need to be modified and it will still run. However, if we want to fully exploit them, they usually make it harder to tune the application.

5 Conclusions

Several advances in the architecture of DSM multiprocessors are in the offing. These advances do not assume technological breakthroughs like, for example, the effective use of superconductor technology for highly-integrated systems. Instead, we based our analysis on known technologies and trends. The advances considered are evolutionary memory hierarchy improvements, processor-memory integration, integration of multiple processors in a chip and speculative parallelization. These advances promise large performance improvements. However, because they enhance the memory hierarchy in non-trivial ways, to truly exploit them will often involve even more efforts in application performance-tuning.

Acknowledgments

The author would like to thank Sujoy Basu, Anthony Nguyen and Ye Zhang for their feedback.

References

[1] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of Existing Architectures in IRAM Systems. In *First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

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

[3] B. Falsafi and D. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[4] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[5] E. Hagersten and M. Koster. WildFire – A Scalable Path for SMPs. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.

[6] IBM Corporation. http://www.chips.ibm.com/ services/foundry/offerings/cmos/7ld. 1998.

[7] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[8] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, pages 88–97, October 1996.

[9] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *Proceedings of the 1998 International Conference on Supercomputing*, July 1998.

[10] A. Moga and M. Dubois. The Effectiveness of SRAM Network Caches in Clustered DSMs. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 103–112, February 1998.

[11] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1996.

[12] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Tomas, and K. Yelick. A Case for Intelligent DRAM. In *IEEE Micro*, pages 33–44, March/April 1997.

[13] D. Patterson and M. Smith. First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. June 1997.

[14] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[15] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 276–285, January 1995.

[16] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors*. SIA, 1997.

[17] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[18] S.Przybylski. Embedded DRAMs: Today and Toward System-Level Integration. Half-day seminar, May 1997.

[19] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[20] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[21] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[22] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[23] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.

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

[25] Z. Zhang and J. Torrellas. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 272–281, February 1997.