Scanning the Issue

Special Issue on Distributed Shared Memory Systems

I. What Is Distributed Shared Memory?

Parallel computer architectures are an important technology to accommodate the performance demands of many emerging applications in areas such as information processing, multimedia, and scientific and technical computing. Research in parallel computer architectures has recently led to an interesting unification of different architectural styles. Distributed shared memory multiprocessors (DSM's) are such a unifying style which aims at combining the best of two existing concepts—shared memory or symmetric multiprocessors (SMP's) and message-passing multicomputers (MMC's).

SMP's and MMC's offer quite different tradeoffs when it comes to ease of programming and scalability, meaning incremental increase in performance when system resources are added. In both styles, multiple high-performance microprocessors are combined in such a way that the computation can be partitioned across them in the hope of achieving a speedup proportional to the number of processors. However, the models SMP's and MMC's expose to the software to support coordination and communication among parallel subcomputations are fundamentally different.

In SMP's, all processors share the same logical address space through a physical shared memory. Because all processors can access any part of a data structure in a parallel execution, the shared memory is used to support coordination and communication. This communication abstraction simplifies parallelization by compilers or programmers and resource utilization at run time. In terms of implementation efficiency, however, a physically shared memory inherently cannot be scaled to large numbers of processors. SMP's thus simplify programming at the expense of limited scalability. MMC's, on the other hand, offer higher scalability by not letting processors share memory physically. The basic abstraction for coordination and communication provided by such systems is instead message exchanges among processors and their private address spaces. Unfortunately, the programmer (or compiler) experiences a more complex model for communication and coordination, which also complicates efficient resource utilization at run time. Thus, MMC's offer a higher scalability at the expense of a more complex programming model.

Publisher Item Identifier S 0018-9219(99)01658-8.

DSM systems aim at combining the ease of programming of SMP's with the scalability of MMC's. They approach this goal by letting each processor have its own physical memory; however, all processors share the same and unique logical address space. Consequently, DSM systems are both scalable, since they do not physically share memory, and they provide a simple programming model by supporting a shared logical address space. While there still remain problems to be solved, viable approaches to design DSM systems have emerged, and this Special Issue summarizes the major research findings behind their emergence.

The basic organization of a DSM system is naturally similar to an MMC. A number of highly tuned compute nodes are connected by a scalable interconnection network that establishes the basic support for efficient message exchanges. The design of the interconnection network is then as important for DSM systems as for MMC systems.

Interconnection networks (IN's) should offer a short latency (traversal time) and a high bandwidth (capacity) at a reasonable cost and with ease of packaging. This cost–performance tradeoff often depends on the number of nodes in the system. In systems with a few (typically tens of) nodes, buses and rings have been popular choices owing to their low cost and latencies. In systems with say 100 nodes, other solutions with higher bandwidths are needed. Crossbars use a grid of buses and offer higher bandwidth and low latencies but become impractical for hundreds of nodes. Because DSM's target hundreds of nodes, IN's for them often trade design complexity for longer latencies by limiting the connectivity to just a few adjacent nodes. This makes their topology and routing strategies critical for the latencies and design cost. Popular examples of such IN's are meshes and tori. Typically, nodes are laid out on a grid and messages are routed from the source to the destination through intermediary nodes. Apart from such specially designed IN's, attractive alternatives from a cost perspective also include commodity local area network (LAN) technologies, such as Ethernet and ATM, which play an important role in DSM's built on top of networks of workstations or personal computers.

Each compute node in a DSM can be a single-processor machine or a full-blown multiprocessor (usually an SMP machine) and consists of processors and their caches and a portion of the system memory. Besides the memory hierarchy in each node, the distributed memory organization adds another level to this memory hierarchy. The ratio of access costs to different levels in this extended memory hierarchy can be several orders of magnitude, making its management performance critical. Whereas the programmer (or compiler) is responsible for managing the additional level in an MMC, the goal is to transparently manage it by hardware or software mechanisms in a DSM system.

II. The Design Space of DSM

The overall goal that DSM system designers face is to provide cost-effective algorithms to manage the extended memory hierarchy so that data can be accessed at a low access cost by still preserving a logically shared address space to the programmer. The design space is naturally huge. In order to put key concepts in this design space into perspective and to provide a background to the articles in this Special Issue, we next review the major options.

Management of the extended memory hierarchy is to a large extent a matter of finding efficient algorithms for moving data dynamically across the different levels (memory or cache levels). One aspect of this is how to map the data structures in the logically shared address space onto the memory modules in the distributed memory organization. Portions of the logical memory space are mapped onto the physical memory, either uniquely (one logical portion mapped to one physical location of the same size), as in cache-coherent nonuniform memory access (NUMA) machines [14], or with replication (one logical portion mapped to several physical locations, each one of the same size as the logical portion), as in cache-only memory architectures (COMA) machines [8] and in reflective memory machines [17]. In general, if the mapping uses replication, mechanisms are needed to maintain the consistency of replicated data. Cache or memory coherence schemes can be implemented in hardware or software or both. These mechanisms can be based either on invalidation or updating of copies. Advanced mechanisms can be hybrid (using invalidation for some data types and update for other data types) or adaptive (using invalidation during some time intervals and update during other time intervals, with an algorithm to switch back and forth) [19]. A key mechanism in a DSM system to aid the coherence scheme is a central bookkeeper, called a directory, that at any point in time knows exactly which nodes have copies of a data item. The exact implementation tradeoffs involved for this mechanism have triggered a lot of research. Typical organizations that have been studied are: 1) full directory; 2) limited directory; and 3) linked-list-based directory. The tradeoff involved is between performance and scalability. For details, see [15] and [16].

Another design parameter is the granularity at which data are moved across the levels in the extended memory hierarchy. It can be an object without semantic meaning (determined via address range) or an object with semantic meaning (determined through data structures). The size of objects without semantic meaning can range from one or a few words to a larger portion of memory, such as a page. Objects with semantic meaning can be a segment, a simple data structure, or a more complex data structure. If the DSM mechanisms are built in hardware, the unit of consistency is typically smaller (a single word or a smaller block of words). One reason why grain size is critical is that it may involve a complex tradeoff between locality and coherence overhead. While larger data chunks can exploit spatial locality and consequently improve the performance of the memory hierarchy, larger chunks may also cause false sharing because accesses from many processors may be directed to different items in the same chunk and can then cause coherence overhead (e.g., [6]). On the other hand, by adopting a large grain size one can amortize the access cost on a larger number of words. This is sometimes a good tradeoff when the DSM mechanism is implemented in software.

The design of the extended memory hierarchy also impacts issues like latency and bandwidth. Of course, these issues are primarily technology dependent; however, concrete values depend on the organization of the memory hierarchy. The needed data can be located in the cache(s) of the same node, in the main memory of the same node, or in another node. If the algorithm for management of the extended memory hierarchy is not perfect, the latencies can be harmful for performance if the processors must wait for the data to be supplied. Ideally, the processors should do useful work while the data are being transferred. This is the key goal of latency-tolerating techniques. Some latency-tolerating techniques are completely transparent to the software, while others make the semantics of the shared-address-space programming model more complex. Examples of the former are prefetching schemes (consumer-initiated data transfers) [15] and data injection schemes (producer-initiated data transfers) [1], [19], while examples of the latter are relaxed memory consistency models, which we return to later.

Another major challenge for DSM designers is to find implementations which are both speed superior and commercially efficient. Obviously, there is a tradeoff between these two issues (the traditional performance/complexity tradeoff). One of these tradeoffs is concerned with whether to implement the DSM algorithms in software, in hardware, or in some combination of the two. Therefore, it is not surprising that all three approaches have been investigated in the research community. Historically, software implementations appeared first (as library routines on the user level, as compiler-level modifications, or as operating system modifications), but they were relatively slow. That is why some researchers turned into hardware implementations, but these were time consuming to implement. Consequently, many have decided that hybrid approaches are the best way to go (some DSM mechanisms implemented in hardware and some in software). As already indicated in the case of the grain size, concrete values of major design parameters depend a lot on the choice of the implementation domain (software, hardware, or hybrid). For more details, see [16] and [17].

The freedom by which the DSM memory hierarchy is managed, be it in hardware or software, is to a large extent limited by the semantics of the shared-address-space programming model. This model views the memory system as if it consisted of one monolithic memory. This monolithic-memory view guarantees that: 1) individual load and store accesses are carried out atomically, i.e., coherence, and 2) a sequence of loads and stores to different locations from the same processor appears as if it were carried out in program order with respect to that processor, i.e., sequential consistency [13]. Another key challenge then is to implement this semantic view with minimal performance losses. Sequential consistency is an example of a memory consistency model (MCM), and elaboration of its implementations and alternative MCM's is currently a major research avenue in the general field of DSM.

Two major types of MCM's are strict (e.g., linearizability and sequential consistency) and relaxed (e.g., processor, weak, release, lazy release, and entry consistency). Linearizability (or strong ordering) typically requires that each memory operation (a load or a store) is performed before the next one can be issued [5], [11]. If the system adheres to this model it also adheres to sequential consistency, although sequential consistency enables some optimizations not possible under strong ordering. Processor consistency allows loads to bypass previously issued stores and thus provides higher performance at the expense of more complex semantics for the program system [2]. Relaxed memory consistency models [5], such as weak ordering, introduce the notion of synchronization variables; in between the synchronization variables, ordering of memory operations is not imposed, while the synchronization variables themselves have to follow the rules of sequential consistency (ordering is imposed both at acquire and release points). The release consistency model [7] is a relaxation of weak ordering in which ordering is only imposed at the release points (execution does not proceed beyond the release point until after all memory operations to shared variables are performed). The lazy release consistency model [12] imposes access ordering only at the next acquire point; this means that only the process that requests exclusive access to data being released must wait. Also, this waiting time is shorter on average, because the next acquire point comes later than the previous release point. Finally, the entry consistency model [4] requires the programmer to introduce new synchronization variables to protect specific data structures or even some important single variables; consequently, the data are passed to the next process only when absolutely needed, and at a later time, which means a potentially better performance but a higher programming effort. From this, it is clear that the more sophisticated the memory consistency model utilized, the more difficult it is to program the underlying machine, but the faster is the code in execution. For more details, see [2].

III. WHY DSM?

The concept of DSM has potential applications in a number of different fields, as evidenced by the articles in the IEEE COMPUTER Special Issue on Applications for Shared-Memory Multiprocessors (December 1996). Some of them are listed and elaborated on here. Others are yet to emerge, as the computing and communications are merging and changing their face. At this time it is obvious that all application domains of DSM listed below will find their niche (more or less wide) in the years to come; however, at this time it is difficult to predict the relative importance that various application domains will get in the years to come.

According to some, shared memory multiprocessing concepts will likely find their way into a next generation of multimicroprocessors on a single chip [10]. After the era of a single microprocessor on a chip is over, one possible next step is to have two microprocessors on a chip, and then more than two on a chip. While the number is about 16 or less, the SMP concept is the obvious choice. However, when the number surpasses about 32 or so, the DSM concept represents a choice which is more suitable. The numbers given here are, of course, technology dependent, and they may change as we see the changes of relative delays of CPU's, buses, and memory.

According to many, the DSM concept is believed to be the underlying architectural concept of the next generation of multimicroprocessor workstations. The ever increasing need for more processing power, which is so dominantly present in the current globalization trends of computing and communications, makes the DSM concept (due to its scalability and programmability) a natural solution for large-scale servers. These days, it is especially true for strategically important Internet servers, but also for servers in other applications traditionally known for their extraordinary large processing power consumption.

The DSM concept is also efficient in combining multiple workstations (or even personal computers) to act as a single system with impressive performance—however, at a much lower cost compared to traditional supercomputers; in this domain of special interest are the experiments which bring DSM into the personal-computer market (e.g., [18])—the approach which is given a special emphasis in the ongoing research oriented to the low-cost markets. Performance issues are supposed to be on the side of the supercomputer approach. (Even if also using the DSM concept, supercomputers can use a number of expensive but efficient technologies for packaging, cooling, and interconnect, which are prohibited for the clusters of workstations or personal computers due to their high cost.) However, the price/performance is definitely on the side of the approaches which utilize the clusters of relatively inexpensive workstations or personal computers.

Thanks to the simple programming model, most major operating systems have been ported to run well on shared memory multiprocessors in general and on DSM systems in particular. The operating system is presumably the most important piece of software and is an important enabler for most applications. The intuitive programming model of DSM systems greatly simplifies operating system implementations, but efficient management of the their memory hierarchies still poses important challenges.

Higher application levels of a typical system can also be supported by DSM. If a DSM concept is built into an operating system it can become better suited for a number of important applications like transaction processing, distributed data engineering, collaboration engineering, and similar applications [21]. Also, DSM may be utilized to bring in the elements of fault tolerance—another dimension of the overall picture which is, unfortunately, frequently neglected and where there are still many important problems to be solved.

IV. Where Do We Go Now?

For a number of years in the past, DSM was a research topic. However, the time has arrived for DSM to start making money, and numerous commercial products are currently on the market with good chances to survive and/or to turn into qualitatively new products with a better market edge. For this reason, an effort has been made in this Special Issue to invite people from industry to prepare a special technology overview. One such technology overview is given by Hagersten and Papadopoulos [9] from Sun Microsystems, Inc., and another one is given by Bell and van Ingen [3] from Microsoft. On the other hand, it is still the academic research which is currently most responsible for moving the state of the art, and papers from leading universities play an important role, both in the real world and in this Special Issue.

The best way to improve the personal awareness of all these research efforts is to consult the proceedings from the major international conferences in the field, e.g., ISCA, HPCA, ASPLOS, and similar. Actually, this is generally an excellent way to create and maintain the awareness of the peak research efforts in this entire field, which is slowly but steadily gaining in its importance.

V. About this Special Issue

This Special Issue contains ten papers that to a great extent summarize the major breakthroughs that have led to the viability of the DSM approach. In "Scanning the Technology," Hagersten and Papadopoulos provide insight into parallel computing in the commercial marketplace. In the next paper, "DSM Perspective: Another Point of View," we invited Bell and van Ingren to help us place the DSM field in greater perspective. The goal set out for the topics of the remaining eight papers is to find an implementation strategy that strikes a good compromise between ease of programming, high performance, and low implementation cost. Two fundamentally different implementation strategies with different market goals in mind have led to two research directions. One is to approach the high-end market, which then allows quite aggressive, predominantly hardware-oriented DSM systems. The first four papers represent this research track. The other direction is to target the low-end market by superimposing a software layer on top of a network of workstations and/or personal computers that implements the DSM paradigm. The last four papers represent this research track.

Whether or not DSM systems offer significant advantages in terms of programmability and scalability as compared to message-passing multicomputers has been an important motivation for many research efforts. Two projects—DASH and Alewife—were launched in the late 1980's to focus on this general issue. The first two papers in this Special Issue summarize the major findings and give perspectives on the lessons learned from these projects. In "Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges," Hennessy *et al.* provide an excellent overview of the technology. The fundamental challenge they faced was to find scalable solutions to the cache-coherence problem. DASH also contributed with several architectural innovations, and their article also discusses them. In "The MIT Alewife Machine," the approach taken by Agarwal *et al.* to integrate shared memory with message passing in the Alewife system is a hybrid of hardware and software techniques. The experience in using these techniques on a large number of applications is the theme of their article.

While the DASH and the Alewife projects demonstrated that the DSM approach is indeed viable, there is still considerable room for improvement. One important approach is to remove the waiting time for the processors by trying to overlap delays in the extended memory hierarchy by computations. Latency-tolerating techniques have gained significant attention, and the next two papers in this issue focus on two important approaches in this direction. In "Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems," Adve *et al.* discuss what optimizations can be enabled under different memory consistency models and how a good compromise can be struck between ease of programming and a high performance. Finally, in "Producer–Consumer Communication in Distributed Shared Memory Multiprocessors" by Byrd and Flynn, the key focus is on techniques to hide the latency inherent to coordination among subcomputations. The approaches they take are to inject speculatively or prefetch data as soon as the value is produced or the consumer knows that it will be needed in the future. They study when the former and latter approaches are beneficial by analyzing their impact on application performance.

Network-connected workstations or personal computers can be converted into a quite powerful DSM system at an attractive cost level if efficient DSM algorithms are incorporated into the software layer. The challenge then lies in finding aggressive software-based policies that can reduce or hide the significant message latencies in such environments. The last four papers summarize state-of-the-art research done in this area and also present prospects for future research. Amza *et al.* implemented the first feasible implementation of a software DSM: the Treadmark system. The key strategies that made it feasible were aggressive use of replication, relaxed memory consistency models, and adaptation to program behavior. In "Adaptive Protocols for Software Distributed Shared Memory," they focus on the implementation of software DSM algorithms and especially how adaptation to program behavior in a program-transparent way can boost the performance of their implementation. Message-passing programming is of course an alternative also on network-connected workstations and/or personal computers. Indeed, compiler technology exists that can convert serial applications to message-passing parallel programs that can provide good performance for problems with predictable and static communication behavior. For problems with less structured and irregular communication behavior, the overheads may severely hamper the performance obtained. Dwarkadas *et al.* have developed a powerful environment in which programs can be compiled and take advantage of the programming model that best matches the application. In "Combining Compile-Time and Run-Time Support for Efficient Software Distributed Shared Memory," they report on their experience in using this environment on a large number of scientific codes. A critical performance aspect of any parallel program implementation is to strike a good compromise between locality and load balance. In "Thread Migration and Communication Minimization in DSM Systems," Thitikamol and Keleher report on the CVM run-time system that aims at minimizing the execution time by migrating the threads in such a way that load imbalance and communication between threads are also minimized. A large number of approaches have been taken to implement software DSM's. The final article, "Shared Virtual Memory: Progress and Challenges" by Iftode and Singh, puts all these ideas into perspective and identifies some important problems that need attention in the future.

This Special Issue gives evidence that the DSM paradigm is maturing; research institutions as well as companies have already demonstrated that the technical challenges can be overcome. At the same time, many open research issues are yet to be addressed. This makes us strongly believe that this technology will increase in its importance in the years to come as demanding applications as well as better implementation technologies will emerge.

REFERENCES

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, V. S. Adve, "An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors," in *Proc. 3rd Int. Symp. High-Performance Computer Architecture,* Feb. 1997, pp. 204–215.

[2] S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Comput.,* vol. 29, pp. 66–76, Dec. 1996.

[3] G. Bell and C. van Ingren, "DSM perspective: Another point of view," this issue, pp. 412–417.

[4] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The midway distributed shared memory system," in *COMPCON'93,* Feb. 1993, pp. 528–537.

[5] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *16th Annu. Int. Symp. Computer Architecture,* 1986, pp. 434–442.

[6] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström, "The detection and elimination of useless misses in multiprocessors," in *20th Annu. Int. Symp. Computer Architecture,* May 1993, pp. 88–97.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *17th Annu. Int. Symp. Computer Architecture,* May 1990, pp. 15–26.

[8] E. Hagersten, A. Landin, and S. Haridi, "DDM—A cache-only memory architecture," *IEEE Comput.,* vol. 25, pp. 44–54, Sept. 1992.

[9] E. Hagersten and G. Papadopoulos, "Parallel computing in the commercial marketplace: Research and innovation at work," this issue, pp. 405–411.

[10] L. Hammond, B. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *IEEE Comput.,* vol. 30, pp. 79–85, Sept. 1997.

[11] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Programming Languages Syst.,* vol. 12, no. 3, pp. 463–492, 1990.

[12] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *19th Annu. Int. Symp. Computer Architecture,* May 1992, pp. 13–21.

[13] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.,* vol. C-28, pp. 241–248, Sept. 1979.

[14] D. Lenoski, J. J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The DASH prototype: Logic overhead and performance," *IEEE Trans. Parallel Distrib. Syst.,* vol. 4, pp. 41–61, Jan. 1993.

[15] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *J. Parallel Distrib. Comput.,* vol. 12, pp. 87–106, July 1991.

[16] J. Protic, M. Tomasevic, and V. Milutinovic, "Survey of distributed shared memory," *IEEE Parallel Distrib. Technol.,* vol. 4, pp. 63–78, Summer 1996.

[17] J. Protic, M. Tomasevic, and V. Milutinovic, *Distributed Shared Memory: Concepts And Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1998.

[18] S. Savic, M. Tomasevic, V. Milutinovic, A. Gupta, M. Natale, and I. Gertner, "Improved RMS for the PC environment," *Microprocessor Syst.,* vol. 19, no. 10, pp. 609–619, Dec. 1995.

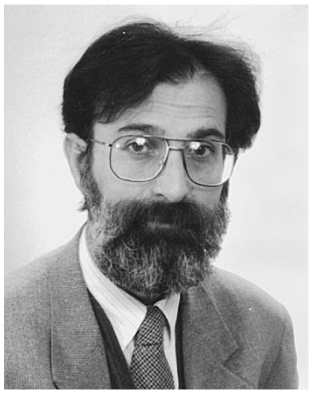
[19] J. Skeppstedt and P. Stenström, "A compiler algorithm that reduces read latency in ownership-based cache coherence protocols," in *Proc. Parallel Architectures and Compilation Techniques,* July 1995, pp. 69–78.

[20] P. Stenström, M. Brorsson, F. Dahlgren, H. Grahn, and M. Dubois, "Boosting performance of shared-memory multiprocessors," *IEEE Comput.,* vol. 30, pp. 63–70, July 1997.

[21] P. Stenström, E. Hagersten, D. Lilja, M. Martonosi, and M. Venugopal, "Trends in shared-memory multiprocessing," *IEEE Comput.,* vol. 30, pp. 44–50, Dec. 1997.

VELJKO MILUTINOVIC, *Guest Editor*
University of Belgrade
Belgrade, Serbia 11120 Yugoslavia

PER STENSTRÖM, *Guest Editor*
Chalmers University of Technology
Gothenburg SE-412 96 Sweden



Veljko Milutinovic *(Senior Member, IEEE) received the Ph.D. degree from the University of Belgrade, Yugoslavia, in 1982.*

Since 1990, he has been a Professor of Computer Engineering with the Department of Computer Engineering, School of Electrical Engineering, University of Belgrade. Previously, he was on the faculty of Purdue University, West Lafayette, IN. His research interests are in computer architecture/design, as well as in system support for electronic business on the Internet. He has contributed more than 50 IEEE journal papers on computer architecture/design and technology-aware system support for mission-critical applications. He has consulted for industry leaders in the United States and Europe, such as IBM, RCA, NCR, VCC, eT, Zycad, Aerospace Corporation, Electrospace Corporation, Intel, Fairchild, Honeywell, Encore, and Phillips. He was the Principal Designer or Project Leader on a number of market successful industrial efforts. He has authored several books and was a coeditor for a number of tutorial books and conference proceedings. He served as Guest Editor for special issues of IEEE COMPUTER *and* IEEE TRANSACTIONS ON COMPUTERS. *He has presented more than 200 invited talks worldwide.*

Dr. Milutinovic is a Fellow of the Serbian Scientific Society of Engineers.

Per Stenström *(Senior Member, IEEE) received the M.S. degree in electrical engineering and the Ph.D. degree in computer engineering from Lund University, Lund, Sweden, in 1981 and 1990, respectively.*

He has been a Professor of Computer Engineering with a Chair in Computer Architecture at Chalmers University of Technology, Gothenburg, Sweden, since 1995. He was previously on the faculty of Lund University. His research interests are in computer architecture and real-time systems, with an emphasis on design principles and design methods for multiprocessor systems, including software as well as hardware design issues. He has contributed more than 50 scientific papers on multiprocessor design principles and authored two textbooks on computer architecture. As a Visiting Scientist, he has participated in major multiprocessor architecture research projects at Carnegie Mellon University, Stanford University, and University of Southern California. He is on the editorial board of the Journal of Parallel and Distributed Computing (JPDC) *and guest edited a special issue of* IEEE COMPUTER *in 1996 on shared memory multiprocessing.*

Dr. Stenström has been a member of numerous program committees on IEEE-sponsored computer architecture and parallel processing conferences. He is a member of the IEEE Computer Society and the ACM.